NASA-CR-200215

IN-61-CR
OCIT
8364
P-99

Final Report

The Detection and Extraction of Interleaved Code Segments

Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills
College of Computing
Georgia Institute of Technology
{spencer, kurt, linda}@cc.gatech.edu

December 20, 1995

NASA Technical Contact: Michael Lowry, NASA Ames Research Center
Research Area: Artificial Intelligence and Software Engineering
Topic Area: Software Understanding and Reengineering
NASA Project Number: NAG 2-890
Georgia Tech Technical Contact:
Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450 Fax: (404) 894-9442

Administrative Contact:
Janis L. Goddard
Contracting Officer
Georgia Tech Research Corporation
Centennial Research Bldg
Georgia Institute of Technology
Atlanta, GA 30332-0420
(404) 894-4817 janis.goddard@oca.gatech.edu

1 Introduction: Interleaving

This project is concerned with a specific difficulty that arises when trying to understand and modify computer programs. In particular, it is concerned with the phenomenon of "interleaving" in which one section of a program accomplishes several purposes, and disentangling the code responsible for each purposes is difficult. Unraveling interleaved code involves discovering the purpose of each strand of computation, as well as understanding why the programmer decided to interleave the strands. Increased understanding improve the productivity and quality of software maintenance, enhancement, and documentation activities.

1.1 Goals

It is the goal of the project to characterize the phenomenon of interleaving as a prerequisite for building tools to detect and extract interleaved code fragments.

1.2 Background

Because the project's approach was largely empirical, we tested it on actual software, the `NAIF SPICELIB` software library obtained from the Jet Propulsion Laboratory. The library consists of 170 thousand lines of Fortran code used by scientists to build software systems to analyze data returned from space missions.

We were introduced to this software by scientists at the NASA Ames laboratory who were familiar with it due to their participation in the Amphion project. It is the purpose of Amphion to improve the productivity of the space scientist by automatically generating software for them, starting from a high-level graphical schematic of a space mission. Amphion is able to do this by using a formal model of the domain (solar system kinematics) and then **proving** that the problem defined by the schematic is derivable from the domain model. In the course of the proof, actual Fortran code is generated which makes appropriate calls to the `SPICELIB` library routines.

The success of Amphion is highly dependent on the consistency and completeness of the domain model it uses, and our role was to use the results we learned about interleaving to improve this model.

2 Summary of Results

The interleaving project was in fact able to accomplish its goals. We have constructed a characterization of interleaving and used the characterization to build detection tools. These tools, in turn, were used to analyze SPICELIB, pointing out parts of the library that the domain model did not cover. The specific analysis routines that we built and the results they obtained are described in this section and included in the Appendices.

Our analyses fall into two categories: those that focus on detecting interleaving-related data from SPICELIB and those that focus on detecting incompletenesses in Amphion's domain model with respect to the library. The results of our empirical analyses on SPICELIB are the following.

- **Precondition extraction** – we have been able to extract preconditions on the use of the SPICELIB routines. It is often difficult to detect the code responsible for checking these preconditions because it is usually tightly interleaved with the code for the primary computation in order to take advantage of intermediate results computed for the primary computation.

  Using the Software Refinery, our tool detects precondition checks within error handlers and extracts the preconditions into a documentation form suitable for expression as a partial specification. The Refine code which implements this detection and extraction is found in Appendix 7.5. A table of preconditions that were extracted is given in Appendix 6. Our tool generates LaTeXoutput so as to easily produce nicely formatted reports. Our tool generated the LaTeXsource included in Appendix 6 without change.

- **Roots of the call graph** – as part of our analysis of the calling structure of the library (which subroutines call others), we have been able to identify which subroutines are not called by any other subroutines. These are listed in Appendix 4.2. Either these represent high-level routines (such as, INELPL) that are intended to be called externally by software that uses SPICELIB, or they are "dead" (no longer used). Perhaps they have been replaced by an optimized or more useful subroutine.

- **Constant parameters** – we have identified calls to library subroutines which supply a parameter to the subroutine that is called as opposed to a variable. The constant parameter may be a flag that is being used to choose among a set of possible computations to perform. The significance of constant parameters is that they often signal a form of interleaving, called *control coupling*. The use of control flags allows control conditions to be determined once but used to affect execution at more than one location in the program. We have discovered that the set of routines that are invoked with

a constant parameter accounts for 19 percent of the total routines in SPICELIB. They are listed in Appendix 4.3.

- **Indefinite loops** – Most of the analyses for detecting interleaving rely on some form of dataflow analysis. Dataflow analysis propagates data usage information along all possible *sequences* of statements in a subprogram. The accuracy or precision of a dataflow analysis depends upon the ability to cover all such sequences. Unfortunately, programs with loops can sometimes represent *indefinite* sequences of statements. When posed with such programs we must settle for only approximate dataflow information. On the other hand, subprograms whose loops are bounded by constants may be *unrolled* into code each of whose statement sequences may be determined statically. Such subprograms can be analyzed exactly with dataflow techniques.

  Since so many of our interleaving analyses depend upon the precision of dataflow analyses, we wanted some measure of our tool's maximum analysis potential over the library. We chose to measure this as the percentage of routines with no indefinite loops and developed a statistics gathering tool in Refine. The analysis showed that roughly 68 percent of the subprograms had no indefinite loops. This number indicates that we will be able to completely analyze two thirds of the routines. Moreover, in the other cases, approximate dataflow information may be good enough to capture the spirit of the analysis.

- **Multiple Outputs and Tupling** – One heuristic for finding instances of interleaving is to determine which subroutines compute more than one output. When this occurs, the subroutine is returning either the results of multiple distinct computations or a result whose type can not be directly expressed in the Fortran type system (e.g., as a data aggregate or tuple). In the former case, the subroutine is realized as the interleaving of multiple distinct plans. In the latter case, the subroutine may be implementing only a single plan, but a maintainer's conceptual categorization of the subroutine is still obscured by the appearance of some number of seemingly distinct outputs.

  We built a tool which uses Refine rules to analyze the direction of dataflow in parameters of SPICELIB functions and subroutines. A parameter's direction is either: **in** if the parameter is only read in the subprogram, **out** if the parameter is only written in the subprogram, or **in-out** if the parameter is both read and written in the subprogram.

  We then analyzed the subroutines in SPICELIB, focusing on the in/out directions of parameters: multiple output subprograms will have more than one parameter with direction out or in-out. Our analysis showed that 25 percent of the subprograms in SPICELIB had multiple output parameters. We were thus able to focus our work on these routines first, as they are likely to involve interleaving.

In addition to analyses for detecting interleaving in the SPICELIB software library, we also performed empirical analyses to determine the coverage provided by the Amphion domain model. In particular, we obtained data on the following.

- **Extent of coverage by routines** – which routines are either directly linked to some entity in the domain model or are invoked by another routine that is covered by the domain model? Based on the calling relationships between library routines and based on Amphion's mapping from domain theory to routines, this analysis revealed that only 35 percent of the library is covered by the domain model.
- **Dead end data flows** – For those routines that are covered by the domain model, which of their outputs are mapped to entities in the domain model? We refer to outputs that are not mapped to anything in the domain model as "dead end dataflows," since the programs that Amphion creates can never make use of these return values; they have not been associated with any meaning in the application domain. Dead end dataflows imply interleaving in the subprogram and/or an incompleteness in the domain model. Our analysis revealed that of the subroutines covered by the domain model, 30 percent have some output parameters that are dead end dataflows.

A paper describing our empirical findings won the best paper award at the International Conference on Software Maintenance this year. We also published our characterization of interleaving at the Working Conference on Reverse Engineering last July. A journal paper on this project has also been submitted to the journal of *Automated Software Engineering* and is currently under review. These papers have been attached to this report.

3 Future Work

There are several possible future directions for pursuing research on interleaving. The suggested directions are summarized here.

3.1 Architectural Interleaving

Thus far, our work has concentrated on detecting instances of interleaving that occur relatively late during design, usually appearing as code optimizations. Significant leverage would obtain if we were able to detect higher level instances of interleaving, specifically situations where several different styles of architecture were interleaved.

There are several promising approaches to this problem. A bottom-up approach tries to detect instances of particular types of modules. For example, the proponents for Structured Design describe module types such as transforming, afferent, efferent, and controlling, all of which can be readily identified due to their dataflow properties. It would be interesting to see the extent to which these types actually occur in their pure form, and, once detected, whether higher level constructs could be detected from them.

Another approach to architectural interleaving detection builds on the concepts of design patterns and frameworks now popular in the object oriented community. It is our belief that characteristic phenomenon of a framework (and possibly of a pattern) is the interaction protocol among its components. We plan to investigate this possibility by instrumenting existing code to produce event traces which can then be analyzed for specific interaction patterns.

The third appealing approach to architectural interleaving investigates the following phenomenon. It turns out that many instances of low level interleaving are really implementations of a more structured design decision, such as aggregation or specialization, made at a higher level in the design. If this is the case, then we might be able to look for explanations of particular patterns of interleaving by detecting the higher level decisions.

3.2 Exploiting Domain Knowledge

Another future direction for research on interleaving exploits domain information. As was the case with `SPICELIB` and Amphion, we expect that our understanding and ability to express domain models and programs will co-evolve. That is, domain models can be constructed or extended by analyzing existing instances of programs in the domain. And as the domain model matures, it will become more and more helpful in understanding programs. One possible application of this will be our ability to express what we learn about a program using a vocabulary that is closer to that of the domain expert than to the programmer.

3.3 Tool Building

The tools that we built for analyzing `SPICELIB` are, for the most part, tailored to it. For example, our detection of preconditions depended on the occurrences of calls to certain error handling routines. It should, however, be possible to build more general tools for detecting and extracting interleavings, based upon the characterization that we have made. Specifically, we need to develop some general tools for doing dataflow analysis in order to refine the approximate analyses that we are currently making. The dataflow tool can then serve as a basis for building tools to detect specific classes of interleavings.

3.4 Further Work with SPICELIB

Finally, there are directions that we could go to further understand SPICELIB. In particular, no tools were developed to detect certain instances of interleaving that we noticed, such as reformulation wrappers. A reformulation wrapper is used to transform one problem into another that is simpler to solve and then to transfer the solution back to the original situation. Some examples of reformulation wrappers in SPICELIB are: reducing a three-dimensional geometry problem to a two-dimensional one and mapping an ellipsoid to the unit sphere to make it easier to solve three-dimensional intersection problems. Once a reformulation wrapper is recognized, properties of the core ("wrapped") computation that are invariant over the reformulation transformation may be associated with the results transformed back to the original problem domain. This is important in recovering accurate post-conditions of SPICELIB subroutines to further elaborate the Amphion domain model. We believe that we could go further in our analysis by looking at some of these cases.

Citation Information for Attached Papers

1. S. Rugaber, K. Stirewalt, and L. Wills. Understanding interleaved code. *Journal of Automated Software Engineering.* Submitted.

2. S. Rugaber, K. Stirewalt, and L. Wills. Detecting interleaving. In *IEEE Conf. on Software Maintenance - 1995*, pages 265–274, Nice, France, September 1995. IEEE Computer Society Press.

3. S. Rugaber, K. Stirewalt, and L. Wills. The interleaving problem in program understanding. In *Proc. of the Second Working Conference on Reverse Engineering*, pages 166–175, Toronto, Ontario, July 1995. IEEE Computer Society Press.

4 Appendix: Empirical Data on NAIF Library

4.1 Preconditions

One of the more ambitious analyses that we perform is the automated detection of subprogram preconditions. Often exception handling behavior can be positively recognized using simple source patterns. We detect precondition checks in code by looking for exception handling code that is invoked under an IF-statement whose predicate consists entirely of unmodified subprogram parameters. When these are discovered, we know that part of the subprogram precondition is the logical negation of this predicate. This general procedure will work for any system whose exception handling mechanism is easily detected. Fortunately, SPICELIB satisfies this condition. The developers followed a strict discipline of exception propagation by registering an exception upon detection and then exiting the executing subprogram. Since their adherence to this policy was so rigorous, we found that we could detect such instances by checking for an invocation of the exception registry procedure SIGERR.

Our precondition extraction code encapsulates the exception recognition code into a single function `looks-like-exception` in the file `exception-pattern.re`. To port the code to a new application, one need merely rewrite this function. Our tool will apply the pattern to any block of code under a conditional whose truth or falsehood depends only upon parameter initial values. As we detect these blocks, we synthesize a partial subprogram precondition as the conjunction of the negation of the predicate of each such conditional. All of this analysis is done by propagating a set of unmodified input variables over the AST of the subprogram looking for exceptional cases. This process uses dataflow propagation techniques from the file `prop-vars.re` and predicate synthesis techniques from the file `precond.re`. (See Appendix 7.5.)

Once we have synthesized the precondition for a given subprogram, we simplify the expression via a predicate optimization pass and then write out the predicate in a LaTeXencoded formula. The accumulated output of all preconditions of all subprograms is then organized into a table via the LaTeXdescription environment. The code that manages this part of the process is in the file `expr-to-latex.re`.

4.2 Roots of the Call Graph

One of our analyses looked at the calling structure of routines in SPICELIB. In the course of this analysis, we were able to identify which subroutines are not called by any other subroutines. These are listed in the table on the next page. Either these represent high-level

routines that are intended to be called externally by software that uses SPICELIB, or they are "dead" (no longer used). Perhaps they have been replaced by an optimized or more useful subroutine.

ALLTRU	APPNDC	APPNDD	APPNDI	APPROX	B1900	B1950
BODEUL	CHBDER	CHGIRF	CKBSR	CKGP	CKGPAV	CKGR01
CKGR02	CKGR03	CKLPF	CKNR01	CKNR03	CKUPF	CKW01
CKW02	CKW03	CLLINE	CLPOOL	CONICS	COPYD	CPOSR
CYCLEC	CYLLAT	CYLREC	CYLSPH	DAFA2B	DAFAH	DAFANA
DAFB2A	DAFBT	DAFCAD	DAFCS	DAFFA	DAFFNH	DAFGH
DAFLUH	DAFNRR	DAFRA	DAFRS	DAFRWD	DAFTB	DATANH
DIFFC	DIFFD	DIFFI	DXTRCT	EDLIMB	ELEMD	ELLTOF
ERRACT	ERRDEV	ERRPRT	ET2UTC	EVEN	EXISTS	FETCHC
FETCHD	FETCHI	FILLC	FILLI	FRELUN	FRSTPC	GEOREC
GETMSG	HYPTOF	INELPL	INRYPL	INSRTD	INTERC	INTERD
INTERI	INVERT	INVSTM	IRFDEF	IRFNAM	ISRCHD	J1900
J1950	J2100	JYEAR	KXTRCT	LATCYL	LATSPH	LBUILD
LDPOOL	LPARSS	LSTCLI	LSTLTC	LSTLTI	M2Q	MATCHW
MAXAC	MAXAD	MAXAI	MEQU	MEQUG	MINAC	MINAD
MTXMG	MTXVG	MXMG	MXMTG	MXVG	NCPOS	NCPOSR
NOTRU	NPEDLN	NPLNPT	OPSGNI	ORDC	ORDD	ORDERC
ORDERI	ORDI	OSCELT	PACKAC	PACKAD	PACKAI	PARTOF
PCWID	POLYDS	POOL	POSR	PRODAD	PRODAI	PROMPT
PRTPKG	QCKTRC	QUOTE	RADREC	RDENCC	RDKER	RECCYL
RECGEO	RECRAD	RECSPH	REMOVD	REMSUB	REORDC	REORDI
REPLWD	REPMCT	REPMF	REPMOT	RESET	RESLUN	ROTVEC
SC01	SCE2S	SCLU01	SCS2E	SDIFFC	SDIFFD	SDIFFI
SETD	SETI	SIGDGT	SMSGNI	SOMTRU	SPCA2B	SPCB2A
SPCDC	SPCOPN	SPCRFL	SPCRNL	SPHCYL	SPHLAT	SPHREC
SPHSD	SPKBSR	SPKEZ	SPKLEF	SPKSUB	SPKUEF	SPKW05
SPKW08	SPKW09	SRFREC	SUMAD	SYDIMC	SYDIMI	SYDUPC
SYDUPD	SYDUPI	SYENQC	SYENQI	SYFETI	SYGETI	SYNTHC
SYNTHI	SYORDD	SYORDI	SYPOPC	SYPOPD	SYPOPI	SYPSHD
SYPSHI	SYPUTC	SYPUTD	SYPUTI	SYRENC	SYREND	SYRENI
SYSELC	SYSELD	SYSELI	SYTRNC	SYTRND	SYTRNI	TIPBOD
TISBOD	TPARCH	TRACEG	TRCOFF	TRCPKG	TYEAR	UNIOND
UNIONI	UNORMG	UTC2ET	VALIDD	VALIDI	VMINUG	VPROJG
VREL	VRELG	VSEPG	VTMV	VTMVG	VUPACK	WNCOMD
WNCOND	WNDIFD	WNELMD	WNEXTD	WNFETD	WNFILD	WNFLTD
WNINTD	WNRELD	WNSUMD	WNUNID	WNVALD	WRENCC	WRPOOL

4.3 Constant Parameters

Subprograms with input parameters that are constant at every call site often indicate the presence of *control coupling.* Routines that exhibit control coupling use the facility to interleave distinct plans with similar dataflow requirements. We want to be able to detect when a routine exhibits control coupling for several reasons:

1. the routine will no doubt be easier to understand if viewed as the interleaving of two distinct plans,

2. the precondition for the routine will probably need to be expressed as the conjunction of several disjoint cases. So, for example, if a routine uses control coupling to interleave two plans S_1 and S_2 with individual preconditions P_1 and P_2 respectively, then we would like the precondition of the routine to be of the form:

$$(CC_1 \Rightarrow P_1) \wedge (CC_2 \Rightarrow P_2)$$

   where CC_1 and CC_2 denote the two disjoint control cases.

3. if we are doing inter-procedural dataflow analysis, we might be able to partially evaluate the routine with the control parameter of interest in order to improve the precision of the analysis.

In Fortran, control coupling is typically implemented by using a subprogram formal parameter as a control flag. We can find possible control flags by finding subprograms having formal parameters whose actuals are constants at *every* call-site in the application. Our analysis shows that 19 percent of the routines in `SPICELIB` are of this form. The routines are listed in the table on the next page.

ACCEPT	ASTRIP	BODVAR	BRCKTD	BRCKTI	BSRCHC	BSRCHD
BSRCHI	CHKIN	CHKOUT	CLEARC	CLEARD	CLEARI	CMPRSS
CONVRT	CPOS	CVPOOL	DAFADA	DAFOPN	DAFPS	DAFRDR
DAFRWA	DAFSIH	DAFT2B	DAFUS	DELTET	DPSTR	DPSTRF
DROTAT	ENCHAR	EQSTR	ERRCH	ERRDP	ERRFNM	ERRINT
ESRCHC	EUL2M	EXACT	EXCESS	FILLD	INSLAC	INSLAD
INSLAI	INSRTC	INSSUB	IOERR	IRFNUM	IRFTRN	ISRCHC
ISROT	LATREC	LPARSE	LPARSM	LSTLTD	M2EUL	MOVEC
MOVED	MSGSEL	NEARPT	NVC2PL	ORDERD	OUTMSG	POS
PREFIX	PUTLMS	PUTSMS	RDENCI	REMLAC	REMLAD	REMLAI
REORDD	REPLCH	REPMC	REPMD	REPMI	REPSUB	ROTATE
RQUAD	RTPOOL	SCARDC	SCARDD	SCLD01	SCLI01	SETC
SETERR	SETMSG	SHIFTC	SHIFTL	SHIFTR	SIGERR	SOMFLS
SPCAC	SSIZEC	SSIZED	SSIZEI	SUFFIX	SUMAI	SWAPAC
SWAPAD	SWAPAI	SWPOOL	TWOVEC	UNITIM	VADDG	VALIDC
VEQUG	VHATG	VLCOM	VLCOM3	VLCOMG	VNORMG	VPACK
VSCLG	VSUBG	VZEROG	WRENCI	WRKVAR	WRLINE	XPOSBL
XPOSEG						

4.4 Indefinite Loops

A major source of analysis imprecision comes from loop statements. Dataflow analysis procedures often can not decide the number of iterations a loop will actually perform. This can come about either because there are branching statements within the loop body or because the loop has an indeterminate upper (or lower) bound. Typically, a dataflow analysis procedure will not distinguish data flow coming in the first iteration of the loop from data flowing from a previous iteration. This makes it difficult to precisely analyze the code within the loop. Loops whose upper and lower bounds are constant, however, may be *unrolled* into straight-forward code barring any unstructured control flow within the loop. Our first analysis discovers all such loops in `SPICELIB` and reports the percentage over all loops. We found that 68 percent of the loops in `SPICELIB` had constant upper and lower bounds. This number gives us confidence that we can statically analyze a large part of the library with high precision.

4.5 Tupling/Pure Routines

Some subroutines in `SPICELIB` compute more than one output. When this occurs, the subroutine is returning either the results of multiple distinct computations or a result whose

type can not be directly expressed in the Fortran type system (e.g., as a data aggregate). In the former case, the subroutine is realized as the interleaving of multiple distinct plans. This interleaving complicates the task of understanding the code, clouds a maintainer's conceptual categorization of the subroutine, and often opens the door to dead-end dataflows (see section 5.2) in the domain theory..

If the result is a type that is not directly expressible in the Fortran type system, then the subroutine may be implementing only a single plan. Still a maintainer's conceptual categorization of the subroutine is obscured by the appearance of some number of seemingly *distinct* outputs. This opens up the potential for a domain theory specialist to *miss* relevant data by thinking that he or she was only interested in one of the outputs when in fact both outputs together determine the value of interest. A good example of this case occurs in the `SPICELIB` subroutine `SURFPT`:

```
SUBROUTINE SURFPT (POSITN,U,A,B,C,POINT,FOUND)
```

which conceptually returns the intersection of a vector with the surface of an ellipsoid. However, it is possible to give a vector and an ellipsoid that do not intersect. In such a situation the output parameter `POINT` will be undefined, but the Fortran type system cannot express the type: `DOUBLE PRECISION` $\vee$ *Undefined*. The programmer was forced to simulate a variable of this type using two variables, `POINT` and `FOUND`, adopting the convention that when `FOUND` is **false**, the return value is *Undefined*, and when `FOUND` is **true**, the return value is `POINT`.

Clearly subprograms with multiple outputs complicate program understanding and can lead to subtle bugs in the domain theory–library mapping. We built a tool that determines the multiple output subprograms in a library by analyzing the *direction* of dataflow in parameters of functions and subroutines. A parameter's direction is either: **in** if the parameter is only read in the subprogram, **out** if the parameter is only written in the subprogram, or **in-out** if the parameter is both read and written in the subprogram. Multiple output subprograms will have more than one parameter with direction out or in-out.

The resulting analysis showed that 25 percent of the subprograms had multiple output parameters. Several of these subprograms are explicitly reachable by sentences in the domain theory, namely:

CKGPAV	CYLLAT	CYLSPH	EL2CGV	INEDPL	INELPL	INRYPL
LATCYL	NEARPT	NPEDLN	NPELPT	NPLNPT	PL2NVC	RECCYL
RECLAT	RECRAD	RECSPH	SCE2T	SPHCYL	SPHLAT	SPKEZ
SPKSSB	SURFPT					

Domain theory specialists should probably check that their understanding of every output of these particular routines is correct. We refer domain theory specialists to the section of domain theory dead-end dataflows 5.2 so that they may check those against this list. Certainly as the domain theory is extended, this problem is more likely to occur. We, therefore, have included the complete list of routines with multiple output parameters:

BODEUL	CHBDER	CHBINT	CHGIRF	CKBSR	CKE01	CKE02
CKE03	CKGP	CKGPAV	CKPFS	CKR01	CKR02	CKR03
CKSNS	COPYC	CYLLAT	CYLSPH	DAFAH	DAFARW	DAFFA
DAFHLU	DAFHSF	DAFLUH	DAFNRR	DAFOPN	DAFPS	DAFRA
DAFRCR	DAFRDA	DAFRDR	DAFRFR	DAFRWA	DAFRWD	DAFWDA
DECHAR	DIAGS2	DIFFC	DP2HX	DXTRCT	EL2CGV	ENCHAR
FNDNWD	FRAME	HX2DP	HX2INT	INEDPL	INELPL	INRYPL
INSLAC	INSLAD	INSLAI	INT2HX	INTERC	IRFNAM	IRFNUM
IRFTRN	KXTRCT	LATCYL	LATSPH	LOCLN	LPARSE	LPARSM
M2EUL	MAXAC	MAXAD	MAXAI	MINAC	MINAD	MINAI
NEARPT	NEXTWD	NPARSD	NPARSI	NPEDLN	NPELPT	NPLNPT
NTHWD	PACKAC	PACKAD	PACKAI	PL2NVC	PL2NVP	PL2PSV
POOL	RAXISA	RDKDAT	RDKER	RDKVAR	RDTEXT	RECCYL
RECGEO	RECLAT	RECRAD	RECSPH	REMLAC	REMLAD	REMLAI
REORDC	REORDD	REORDI	RMDUPC	RMDUPD	RMDUPI	RQUAD
SAELGV	SC01	SCARDC	SCE2S	SCE2T	SCENCD	SCET01
SCFM01	SCFMT	SCLI01	SCLU01	SCPART	SCS2E	SCT2E
SCTE01	SCTIKS	SCTK01	SDIFFC	SETC	SPCRFL	SPCRNL
SPHCYL	SPHLAT	SPKAPP	SPKBSR	SPKE08	SPKE09	SPKEZ
SPKGEO	SPKPV	SPKSFS	SPKSSB	SSIZEC	STMP03	SURFPT
SWAPC	SWAPD	SWAPI	SYDELC	SYDELD	SYDELI	SYDUPC
SYDUPD	SYDUPI	SYENQC	SYENQD	SYENQI	SYFETC	SYFETD
SYFETI	SYGETC	SYGETD	SYGETI	SYNTHC	SYNTHD	SYNTHI
SYORDC	SYORDD	SYORDI	SYPOPC	SYPOPD	SYPOPI	SYPSHC
SYPSHD	SYPSHI	SYPUTC	SYPUTD	SYPUTI	SYRENC	SYREND
SYRENI	SYSELC	SYSELD	SYSELI	SYSETC	SYSETD	SYSETI
SYTRNC	SYTRND	SYTRNI	TIPBOD	TISBOD	TPARSE	TRCPKG
UNIONC	UNORM	UNORMG	VALIDC	VPRJPI	VUPACK	WNDIFD
WNFETD	WNSUMD					

5 Appendix: Empirical Data on Amphion Domain Model

Whenever a declarative system is used to generate code over a library, one would like to know whether or not the library is *covered* by the declarative domain. That is:

1. Can every routine in the library be invoked by *some* sentence in the domain theory, and
2. is every value returned by a routine in the library accounted for in the domain theory?

If the answer to either question is no, then either the library is superfluous or the domain theory is incomplete. We would like to know about either case.

5.1 Extent of Coverage by Domain Model

The first question is one of *coverage.* It can be rephrased as, "What percentage of the routines in the library are covered by the domain model?" Our analyses show that only 35 percent of the library is covered by the domain model.

5.2 Dead End Dataflows

Some routines that are covered by the domain model have outputs that are not mapped to anything in the domain model. We call these outputs "dead end dataflows," since the programs that Amphion creates can never make use of these return values; they have not been associated with any meaning in the application domain. Dead end dataflows imply interleaving in the subprogram and/or an incompleteness in the domain model. Our analysis revealed that of the subroutines covered by the domain model, 30 percent have some output parameters that are dead end dataflows. While is probably not surprising considering that many of the subprograms referenced in the domain model have multiple outputs (see section ??), domain theory specialists should double check that they understand *why* these dead end dataflows exist. That is, is the information returned in these extraneous outputs superfluous? Do they represent values that were merely *convenient* to compute by the library subprogram but that are not pertinent to the computation of interest? If the answer to these questions is no, then the domain theory specialist should consult with a `SPICELIB` specialist to determine the *true* nature of the dataflows.

The dead end dataflows that we found follow:

CKGPAV contains four dead ends: `FOUND CLKOUT AV REF`

PL2NVC contains one dead end: `CONST`

INELPL contains two dead ends: `XPT2 NXPTS`

INEDPL contains one dead end: `FOUND`

NPEDLN contains one dead end: `DIST`

NPELPT contains one dead end: `DIST`

SCE2T contains one dead end: `ET`

NEARPT contains one dead end: `ALT`

INRYPL contains one dead end: `NXPTS`

NPLNPT contains one dead end: `DIST`

SPKSSB contains one dead end: `REF`

SPKEZ contains one dead end: `REF`

RECRAD contains one dead end: `RANGE`

6 Appendix: Preconditions Extracted

What follows is a table of subprogram preconditions indexed by subprogram name. The table was automatically generated from the SPICELIB source code using our precondition extraction code.

CARDD $\neg(INT(CELL(-1)) < 0) \wedge \neg(INT(CELL(0)) > INT(CELL(-1))) \wedge \neg(INT(CELL(0)) < 0)$

CARDI $\neg(CELL(-1) < 0) \wedge \neg(CELL(0) > CELL(-1)) \wedge \neg(CELL(0) < 0)$

CKW01 $\neg(SCLKDP(1) < 0.D0) \wedge \neg(ENDTIM < SCLKDP(NREC)) \wedge \neg(BEGTIM > SCLKDP(1)) \wedge \neg(NREC \leq 0)$

CKW02 $\neg(STOP(1) \leq START(1)) \wedge \neg(START(1) < 0.D0) \wedge \neg(ENDTIM < STOP(NREC)) \wedge \neg(BEGTIM > START(1)) \wedge \neg(NREC \leq 0)$

CKW03 $\neg(SCLKDP(1) < 0.D0) \wedge \neg(ENDTIM < SCLKDP(NREC)) \wedge \neg(BEGTIM > SCLKDP(1)) \wedge \neg(NINTS \leq 0) \wedge \neg(NREC \leq 0)$

COUNTC $\neg((BLINE > ELINE) \vee (BLINE \leq 0))$

CYCLAC $(DIR =' F') \vee (DIR =' f')$

CYCLAD $(DIR =' F') \vee (DIR =' F')$

CYCLAI $(DIR =' F') \vee (DIR =' F')$

CYCLEC $(DIR =' R') \vee (DIR =' r')$

DACOSH $\neg(X < 1.D0)$

DAFAH $\neg(FNAME =' ')$

DAFRA $\neg\neg ISORDV(IORDER, N)$

DAFRDA $\neg(BEGIN \leq 0) \wedge \neg(BEGIN > END)$

DAFRWA $\neg((RECNO \leq 0) \vee (WORDNO \leq 0))$

DAFWDA $\neg(BEGIN \leq 0) \wedge \neg(BEGIN > END)$

DATANH $\neg(DABS(X) \geq 1.0D0)$

DROTAT $\neg((IAXIS > 3) \vee (IAXIS < 1))$

EDLIMB $\neg((A \leq 0.D0) \vee (B \leq 0.D0) \vee (C \leq 0.D0))$

ELLTOF $\neg((ECC < 0.D0) \vee (ECC \geq 1.D0))$

ENCHAR $\neg(NUMBER < 0)$

GEOREC $\neg(F \geq 1) \wedge \neg(RE \leq 0.0D0)$

HYPTOF $\neg(ECC < 1.D0)$

INEDPL $\neg((A \leq 0.D0) \vee (B \leq 0.D0) \vee (C \leq 0.D0))$

INSSUB $\neg((LOC < 1) \vee (LOC > (LEN(IN) + 1)))$

ISROT $\neg(NTOL < 0.D0) \wedge \neg(DTOL < 0.D0)$

LGRESP $\neg(STEP = 0.D0) \wedge \neg(N < 1)$

LGRINT $\neg(N < 1)$

M2EUL $\neg((AXIS3 < 1) \vee (AXIS3 > 3) \vee (AXIS2 < 1) \vee (AXIS2 > 3) \vee (AXIS1 < 1) \vee (AXIS1 > 3)) \wedge \neg((AXIS3 = AXIS2) \vee (AXIS1 = AXIS2))$

NPEDLN $\neg((A \leq 0.D0) \vee (B \leq 0.D0) \vee (C \leq 0.D0))$

NPLNPT $\neg VZERO(LINDIR)$

NVP2PL $\neg VZERO(NORMAL)$

PROP2B $\neg(DT < 0) \wedge \neg(DT > 0)$

RDENCC $\neg(N < 1)$

RDENCD $\neg(N < 1)$

RDENCI $\neg(N < 1)$

RECGEO $\neg(F \geq 1) \wedge \neg(RE \leq 0.0D0)$

REMLAC $\neg((LOC < 1) \vee (LOC > NA)) \wedge \neg(NE > (NA - LOC + 1))$

REMLAD $\neg((LOC < 1) \vee (LOC > NA)) \wedge \neg(NE > (NA - LOC + 1))$

REMLAI $\neg((LOC < 1) \vee (LOC > NA)) \wedge \neg(NE > (NA - LOC + 1))$

REMSUB $\neg((LEFT > RIGHT) \vee (RIGHT < 1) \vee (LEFT < 1) \vee (RIGHT > LEN(IN)) \vee (LEFT > LEN(IN)))$

REPSUB $\neg(LEFT < 1) \wedge \neg(RIGHT < (LEFT - 1))$

RQUAD $\neg((A = 0.D0) \wedge (B = 0.D0))$

SC01 $\neg(CLKSTR = ' ')$

SCARDD $\neg((CARD < 0) \vee (CARD > INT(CELL(-1))))$

SCARDI $\neg((CARD < 0) \vee (CARD > CELL(-1)))$

SCE2T $SCTYPE(SC) = 1$

SCENCD $\neg(POS > 1) \wedge \neg(POS = 1)$

SCT2E $SCTYPE(SC) = 1$

SETC $OP = ' '$

SETD $OP = ' '$

SETI $OP = ' '$

SIZED $\neg(INT(CELL(-1)) < 0) \wedge \neg(INT(CELL(0)) > INT(CELL(-1))) \wedge \neg(INT(CELL(0)) < 0)$

SIZEI $\neg(CELL(-1) < 0) \wedge \neg(CELL(0) > CELL(-1)) \wedge \neg(CELL(0) < 0)$

SPHSD $\neg(RADIUS < 0)$

SPKW05 $\neg(FIRST > LAST) \wedge \neg(N \leq 0)$

SPKW08 $\neg(EPOCH1 > FIRST) \wedge \neg((EPOCH1 + (N-1) * STEP) < LAST) \wedge \neg(STEP \leq 0.D0) \wedge \neg(FIRST \geq LAST) \wedge \neg(N \leq DEGREE)$

SPKW09 $\neg(EPOCHS(1) > FIRST) \wedge \neg(EPOCHS(N) < LAST) \wedge \neg(N \leq DEGREE) \wedge \neg(FIRST \geq LAST)$

SSIZEC $\neg(SIZE < 0)$

SSIZED $\neg(SIZE < 0)$

SSIZEI $\neg(SIZE < 0)$

SURFPT $\neg((U(1) = 0.0D0) \wedge (U(2) = 0.0D0) \wedge (U(3) = 0.0D0))$

SWAPAC $\neg(N < 0) \wedge \neg(LOCN < 1) \wedge \neg(LOCM < 1) \wedge \neg(M < 0)$

SWAPAD $\neg(N < 0) \wedge \neg(LOCN < 1) \wedge \neg(LOCM < 1) \wedge \neg(M < 0)$

SWAPAI $\neg(N < 0) \wedge \neg(LOCN < 1) \wedge \neg(LOCM < 1) \wedge \neg(M < 0)$

SYPUTC $\neg(N < 1)$

SYPUTD $\neg(N < 1)$

SYPUTI $\neg(N < 1)$

TWOVEC $\neg((MAX(INDEXP, INDEXA) > 3) \vee (MIN(INDEXP, INDEXA) < 1)) \wedge \neg(INDEXA = INDEXP)$

TXTOPN $\neg(FNAME = ' ')$

TXTOPR $\neg(FNAME = ' ')$

VALIDC $\neg(N > SIZE)$

VALIDD $\neg(N > SIZE)$

VALIDI $\neg(N > SIZE)$

WNCOMD $\neg(LEFT > RIGHT)$

WNINSD $\neg(LEFT > RIGHT)$

WNRELD $(OP = '>=') \vee (OP = '>')$

WNVALD $\neg ODD(N) \wedge \neg(N > SIZE)$

WRENCC $\neg(N < 1)$

WRENCD $\neg(N < 1)$

WRENCI $\neg(N < 1)$

XPOSBL $\neg((MOD(NCOL, BSIZE) \neq 0) \vee (MOD(NROW, BSIZE) \neq 0)) \wedge \neg(NCOL < 1) \wedge \neg(NROW < 1) \wedge \neg(BSIZE < 1)$

7 Appendix: Refine Code

7.1 The Precondition Extraction Code

The code to perform the precondition extraction follows. Note that the file `exception-pattern.re` is specialized to the `SPICELIB` but is easily ported to other applications.

The file exception-pattern.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  This file encapsulates the application dependent exception handling
  recognition code. This particular instance is specialized to detecting
  exceptions in SPICELIB
||#
var *FOUND-RETURN*       : boolean = undefined
var *FOUND-SIGERR*          : boolean = undefined
rule IS-RETURN ( s : rf::program-unit-statement )
   ~*FOUND-RETURN* &
   rf::return-statement(s)
   -->
   *FOUND-RETURN*
rule IS-SIGERR ( s : rf::program-unit-statement )
   ~*FOUND-SIGERR* &
   rf::call-statement(s) &
   rf::identifier-name(rf::called-object(s)) =
                                  'RFU::SIGERR
   -->
   *FOUND-SIGERR*
"The rules that check for patterns indicating the application is raising
 an exception. We found that the JPL programmers used a very nice scheme
 for handling exceptions, and that a block of code calling the routine
 SIGERR followed by a RETURN statement denotes exception handling."
var *CHECK-EXCEPTION-RULES* : seq(symbol) =
  [ 'IS-RETURN,
    'IS-SIGERR ]
"Given a sequence of Fortran statements (assumed to be the body of an
 IF-THEN-ELSE block), report whether or not it appears to be raising
 an exception."
```

```
function looks-like-exception( stmt-seq : seq(rf::program-unit-statement) ) :
   boolean =
   let (*FOUND-RETURN*       : boolean = false,
        *FOUND-SIGERR* : boolean = false)
      (enumerate a-stmt over stmt-seq do
          preorder-transform(a-stmt, *CHECK-EXCEPTION-RULES*));
      *FOUND-RETURN* & *FOUND-SIGERR*
```

The file prop-vars.re

```
!! in-package("RU")
!! in-grammar('user)
#||
   This file contains code to propagate a list of variables which have not
   been modified through operations which may modify them. The outcome
   of this analysis is a list of variables *still* not modified after
   passing through the given operations.
   Entry Point:
      Propagate-Vars-Through-Expression(expression, set of variables)
      returns set of variables
||#
var *ANALYSIS-SC* : rlt::structure-chart = undefined
function find-in-structure-chart( i : rf::identifier-or-sref ) : rlt::sc-node =
  if(rf::identifier(i))
  then some (sc)(sc in rlt::sc-nodes(*ANALYSIS-SC*) &
                rlt::sc-node-name(sc) = symbol-to-string(
                                                    rf::identifier-name(i)))
  else undefined
function compare-by-position(params   : seq(rf::expression-or-special),
                             idparams : set( rf::expression-or-special ),
                             sc-node  : rlt::sc-node ) : set(symbol) =
  let (out-formals : seq( rlt::sc-formal ) =
                   [ fp | (fp) fp in rlt::sc-node-formals(sc-node) &
                                rlt::sc-formal-direction(fp) ~= 'rlt::in ] )
     let (modifiable-actuals : set(rf::expression-or-special) =
           ac | (ac,fp,i) fp in out-formals &
                          ac in idparams &
                          i in [1..size(rlt::sc-node-formals(sc-node))] &
                          fp = rlt::sc-node-formals(sc-node)(i) &
                          ac = params(i)  )
        image(lambda (a : rf::expression-or-special)
                    let ( a : rf::identifier-or-sref =
                              rf::id-or-indexee(a))
                       if (rf::identifier(a)) then rf::identifier-name(a)
                       else undefined ,
              modifiable-actuals)
"Given a list of parameters to a function or CALL statement, return the
 subset of this list which are simple variable names as opposed to more
```

```
 complex expressions."
function get-identifier-parameters( el : seq( rf::expression-or-special ) ) :
  set(rf::expression-or-special) =
   i | (i : rf::expression-or-special)
         i in el &
         rf::indexable-name(i) &
         ~rf::indexable-name-has-params?(i)
"Given a structure chart node sc-node which represents a called function
 or subroutine, a sequence params of actual parameters to this call, and
 a set vars of immutable variable names, return the subset of vars that
 is still immutable after the execution of the call."
function safe-vars-through-call(sc-node : rlt::sc-node,
                                params  : seq(rf::expression-or-special),
                                vars    : set(symbol) ) :
       set(symbol) =
  let ( ident-params : set( rf::expression-or-special ) =
                       get-identifier-parameters( params ) )
     let ( expr-params : set(rf::expression-or-special) =
                         setdiff(seq-to-set(params),
                                 ident-params))
        let ( newvars : set(symbol) =
                        if defined?(sc-node)
                        then setdiff(vars, compare-by-position(params,
                                                               ident-params,
                                                               sc-node))
                        else vars)
           if empty(expr-params) then
             newvars
           else reduce(intersect,
                       image(lambda (e : rf::expression-or-special)
                             check-expression-or-special(e,
                                                         newvars),
                             expr-params))
#||
  Rationale: Given something that looks like an array ref or a function call,
    there are two cases in which variables can be modified:
       1) The actual parameter is a variable, and the entity is a function
          which modifies it's formal parameter of this position, or
       2) The actual parameter is an expression which itself is a function
          call that modifies parameters.
```

```
  This recursive definition allows us to manage the problem by:
     1) Split the actual parameter list into 2 sets, one of which is
        lone variables, and the other expressions.
     2) Go look up the suspected subprogram in the structure chart. If not
        found, then it was an array ref and we're done with set 1, else
        it was a function call, so we match actuals in set 1 with formals
        in the structure chart node and report cases which can be modified.
     3) Apply a recursive call to check-expression to each member
        of set 2 getting a set of sets of modified vars. Reduce this set
        by append to get a flattened set.
     4) Return the union of the sets output in steps 2 and 3.
|#
function check-indexable-name( fc   : rf::indexable-name,
                               vars : set(symbol) ) :
  set(symbol) =
  if (~rf::indexable-name-has-params?(fc)) then
     vars
  else let( params : seq(rf::expression-or-special) =
                          rf::function-params-or-indices(fc),
            sc-info : rlt::sc-node = find-in-structure-chart(
                                                  rf::id-or-indexee(fc)))
          if defined?(sc-info)
          then safe-vars-through-call(sc-info, params, vars)
          else vars
function check-expression( e    : rf::expression,
                           vars : set(symbol) ) : set(symbol)
  computed-using
    rf::indexable-name(e) => check-expression(e,vars) =
                                 check-indexable-name(e,vars),
    rf::binary-expression(e) => check-expression(e,vars) =
                                 reduce(intersect,
                                        image(lambda (x)
                                                 check-expression(x,vars),
                                              rf::exp-list(e))),
    rf::unary-expression(e) => check-expression(e,vars) =
                                 check-expression(rf::exp-value(e), vars),
    rf::literal-constant(e) or
    rf::open-key-parameter(e) or
    rf::special-intrinsic-call(e) => check-expression(e,vars) = vars
function check-expression-or-special( e    : rf::expression-or-special,
```

```
                                        vars : set(symbol) ) :
  set(symbol)
  computed-using
     rf::expression(e) => check-expression-or-special(e,vars) =
                          check-expression(e,vars),
     rf::label-parameter(e) or rf::null-ex(e) =>
                     check-expression-or-special(e,vars) = vars
"Given an expression e and a set vars of immutable variables, return the
 subset of vars which are still immutable."
function Propagate-Vars-Through-Expression( e    : rf::expression-or-special,
                                            vars : set(symbol) ) :
     set(symbol) =
     check-expression-or-special(e,vars)
function get-vars-from-indexable-name( fc   : rf::indexable-name ) :
  set(symbol) =
  let( sc-info : rlt::sc-node = find-in-structure-chart(rf::id-or-indexee(fc)),
       flat-pars : set(symbol) =
                   (if (rf::indexable-name-has-params?(fc))
                    then let( params : seq(rf::expression-or-special) =
                           rf::function-params-or-indices(fc) )
                         reduce(union, image(get-vars-from-expr, params))
                    else ))
  if(defined?(sc-info))
  then flat-pars
  else flat-pars with rf::identifier-name(rf::id-or-indexee(fc))
function get-vars-from-expr( e    : rf::expression ) : set(symbol)
  computed-using
    rf::indexable-name(e) => get-vars-from-expr(e) =
                                 get-vars-from-indexable-name(e),
    rf::binary-expression(e) => get-vars-from-expr(e) =
                                reduce(union,
                                       image(get-vars-from-expr,
                                             rf::exp-list(e))),
    rf::unary-expression(e) => get-vars-from-expr(e) =
                               get-vars-from-expr(rf::exp-value(e)),
    rf::literal-constant(e) or
    rf::open-key-parameter(e) or
    rf::special-intrinsic-call(e) => get-vars-from-expr(e) =
function get-vars-from-expr-or-special( e    : rf::expression-or-special ) :
  set(symbol)
```

```
computed-using
   rf::expression(e) => get-vars-from-expr-or-special(e) =
                        get-vars-from-expr(e),
   rf::label-parameter(e) or rf::null-ex(e) =>
                   get-vars-from-expr-or-special(e) =
```

The file precond.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  Entry Point:
      compute-preconditions( this-sys : rf::system-analysis ) :
        string
  Returns:
      Description of all detectable  preconditions (detected by explicit
        checks whose failure causes an exception). The description is
        in the form of a LaTeX description environment.
||#
type condition-tuple-type = tuple(rf::expression-or-special,symbol)
type condition-tuple-set = set(condition-tuple-type)
"Given a system-analysis, return the cumulative sequence of subroutines
 that are contained in this analysis."
function convert-analysis-into-subr-seq( this-system : rf::system-analysis ) :
  seq(rf::program-unit) =
  analysis::maybe-wrapup-system-analysis(this-system);
  let (exec-progs : seq(rf::executable-program) =
          [ e | (file-an : rf::file-analysis,
                 e       : rf::executable-program)
                 file-an in rf::file-analyses-for-system(this-system) &
                 e = rf::file-analysis-file-ast(file-an) ])
     reduce(append, [ s | ( p : rf::executable-program,
                            s : seq(rf::program-unit) )
                          p in exec-progs &
                          s = rf::program-units(p) &
                          defined?(s) &
                          ~empty(s) ] )
#||
  EXPLICIT DATAFLOW PROPAGATION ROUTINES.
||#
function prop-assign-statement( stmt  : rf::assignment-statement,
                                vars  : set(symbol) ) :
                 set(symbol) =
  let (lhs : rf::indexable-name = rf::assignee(stmt))
     let (simple-lhs : symbol = rf::identifier-name(rf::id-or-indexee(lhs)))
        let (complex-lhs : set(symbol) =
```

```
                        if (rf::indexable-name-has-params?(lhs) and-then
                            defined?(rf::function-params-or-indices(lhs))
                            and-then
                            ~empty(rf::function-params-or-indices(lhs)))
                        then reduce(intersect,
                                    image(lambda (e)
                                       Propagate-Vars-Through-Expression(e,
                                                                         vars),
                                       rf::function-params-or-indices(lhs)))
                        else vars)
          let( rhs : rf::expression-or-special = rf::exp-value(stmt) )
             let (after-exc  : set(symbol) =
                  intersect( Propagate-Vars-Through-Expression(rhs,
                                                               vars),
                             complex-lhs)
                  less simple-lhs )
               after-exc
function prop-call( stmt : rf::call-statement,
                    vars : set(symbol) ) : set(symbol) =
  let (call-pars  : seq(rf::expression-or-special) = rf::call-params(stmt),
       sc-node    : rlt::sc-node = find-in-structure-chart(
                                          rf::called-object(stmt)))
  safe-vars-through-call(sc-node,
                         call-pars,
                         vars)
function run-through-stmt-seq(
                       stmts : seq(rf::program-unit-statement),
                       vars  : set(symbol)) : set(symbol) =
  let (newvars : set(symbol) = vars)
    (enumerate s over stmts do
       newvars <- Propagate-Vars-Through-Executable-Statement(s,newvars));
    newvars
function prop-do( stmt : rf::do-statement,
                  vars : set(symbol) ) : set(symbol) =
  let (do-stms : seq(rf::program-unit-non-enddo-statement) =
                 rf::do-actions(stmt))
     run-through-stmt-seq(do-stms, vars)
function prop-arith-if( stmt : rf::arithmetic-if-statement,
                        vars : set(symbol) ) : set(symbol) =
  Propagate-Vars-Through-Expression( rf::exp-value(stmt), vars )
```

```
function prop-logical-if-statement( stmt : rf::logical-if-statement,
                                    vars : set(symbol) ) :
             set(symbol) =
  Propagate-Vars-Through-Executable-Statement(
           rf::logical-if-stmt(stmt),
           Propagate-Vars-Through-Expression(
                rf::exp-value(stmt),
                vars))
function prop-if-then-else( stmt : rf::if-then-else-statement,
                            vars : set(symbol) ) :
             set(symbol) =
  let (exp-val : rf::expression-or-special = rf::exp-value(stmt))
  let ( exp-vars : set(symbol) = Propagate-Vars-Through-Expression(
                                              exp-val, vars))
     let ( then-vars : set(symbol) =
                       run-through-stmt-seq(rf::then-part(stmt), exp-vars))
        let( else-vars : set(symbol) =
                         if defined?(rf::else-part(stmt)) then
                            run-through-stmt-seq(rf::else-part(stmt), exp-vars)
                         else then-vars )
           let( elsif-vars : set(symbol) =
                             if defined?(rf::elseif-part(stmt)) then
                                prop-if-then-else( rf::elseif-part(stmt),
                                                   exp-vars )
                             else else-vars )
              intersect(then-vars, intersect(else-vars, elsif-vars))
function Propagate-Vars-Through-Executable-Statement(
                                    stmt : rf::executable-statement,
                                    vars : set( symbol ) ) :
                          set( symbol )
  computed-using
  rf::assignment-statement(stmt) =>
                  Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                       prop-assign-statement(stmt,vars),
  rf::call-statement(stmt) =>
                  Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                       prop-call(stmt,vars),
  rf::arithmetic-if-statement(stmt) =>
                  Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                       prop-arith-if(stmt, vars),
```

```
  rf::block-if-statement(stmt) =>
                    Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                         prop-if-then-else(rf::conditional-if(stmt),
                                           vars),
  rf::logical-if-statement(stmt) =>
                    Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                         prop-logical-if-statement(stmt, vars),
  rf::do-statement(stmt) =>
                    Propagate-Vars-Through-Executable-Statement(stmt,vars) =
                         prop-do(stmt,vars),
%%
%% VERY conservative dataflow here. We could do better.
%%
  rf::include-statement(stmt) or
  rf::io-statement(stmt) => Propagate-Vars-Through-Executable-Statement(
                                                          stmt, vars ) = ,
%%
%% These statements can not affect mutation of variables.
%%
  rf::declaration-statement(stmt) or
  rf::entry-statement(stmt) or
  rf::format-statement(stmt) or
  rf::label-definition(stmt) or
  rf::continue-statement(stmt) or
  rf::end-do-statement(stmt) or
  rf::goto-statement(stmt) or
  rf::label-assignment-statement(stmt) or
  rf::pause-statement(stmt) or
  rf::return-statement(stmt) or
  rf::save-statement(stmt) or
  rf::stop-statement(stmt) =>
           Propagate-Vars-Through-Executable-Statement(stmt,vars) = vars
function depends-only-upon-vars( e    : rf::expression-or-special,
                                 vars : set(symbol) ) : boolean =
  let(used-vars : set(symbol) = get-vars-from-expr-or-special(e))
     used-vars subset vars
#||
  PATTERN MATCHING CODE FOLLOWS
||#
function block-if-matches(the-if : rf::if-then-else-statement,
```

```
                          vars    : set(symbol)) :
  condition-tuple-set =
  let (exp-val : rf::expression-or-special = rf::exp-value(the-if))
  if (defined?(the-if))
  then union(
            (if (depends-only-upon-vars(exp-val,vars))
             then (if looks-like-exception(rf::then-part(the-if))
                   then  <exp-val,'NEGATIVE>
                   elseif defined?(rf::else-part(the-if)) &
                          looks-like-exception(rf::else-part(the-if))
                   then  <exp-val,'POSITIVE>
                   else )
             else ),
            (if defined?(rf::elseif-part(the-if))
             then block-if-matches(rf::elseif-part(the-if), vars)
             else  ))
  else
"At the present, logical if's can not match as precondition checks because
 they can not execute both a call to signal and exception and a return."
function logical-if-matches(s     : rf::logical-if-statement,
                            vars : set(symbol)) :
        condition-tuple-set =
function get-vars-from-subroutine( pu : rf::program-unit )  : string =
  let ( vars : set(symbol) =  rf::identifier-name(
                                      rf::id-name(rf::parameter-name(p))) |
                                             (p) p in rf::formals(
                                                    rf::unit-heading(pu)) )
    let (newvars    : set(symbol) = vars,
         candidates : condition-tuple-set = )
       (enumerate s over rf::unit-body(pu) do
           (if rf::logical-if-statement(s)
            then candidates <- union(candidates,
                                     logical-if-matches(s,newvars))
            elseif rf::block-if-statement(s)
                then candidates <- union(candidates,
                                         block-if-matches(
                                            rf::conditional-if(s),newvars)));
          newvars <- Propagate-Vars-Through-Executable-Statement(s,newvars));
       if (empty(candidates))
       then ""
```

```
        else let(seed : condition-tuple-type = arb(candidates))
               let (out-str : string =
                           precond-to-latex( seed.1,
                                             (seed.2 = 'NEGATIVE)))
               (enumerate i over (candidates less seed) do
                  out-str <- concat(out-str,
                                    " \wedge ",
                                    precond-to-latex(
                                             i.1,
                                             (i.2 = 'NEGATIVE))));
              out-str
"Main entry point. Given a system-analysis, computes the preconditions
 and displays them in a LaTeX suitable form."
function compute-preconditions( this-sys : rf::system-analysis ) =
       let( *ANALYSIS-SC* : rlt::structure-chart =
              fret::extract-structure-chart-from-symbol-table(this-sys))
        let (program-units : seq(rf::program-unit) =
                           convert-analysis-into-subr-seq(this-sys))
          format(true, "\begindescription~%");
          (enumerate p over program-units do
             let ( pred-str : string = get-vars-from-subroutine(p))
               if (~empty(pred-str))
               then format(true, "\item[");
                    format(true,
                           symbol-to-string(
                              rf::identifier-name(
                                 rf::id-name(rf::unit-heading(p)))));
                    format(true, "] $");
                    format(true, pred-str);
                    format(true, "$~%"));
          format(true, "\enddescription~%")
"A specialization of compute-preconditions to the NAIF library"
function naif-preconditions () =
    let(system-name : string =
          "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis")
     let (this-sys : rf::system-analysis = analysis::load-system-analysis(
                                                          system-name))
       analysis::maybe-wrapup-system-analysis(this-sys);
       compute-preconditions(this-sys)
```

```
        else let(seed : condition-tuple-type = arb(candidates))
               let (out-str : string =
                           precond-to-latex( seed.1,
                                             (seed.2 = 'NEGATIVE)))
                (enumerate i over (candidates less seed) do
                   out-str <- concat(out-str,
                                     " \wedge ",
                                     precond-to-latex(
                                              i.1,
                                              (i.2 = 'NEGATIVE))));
               out-str
"Main entry point. Given a system-analysis, computes the preconditions
 and displays them in a LaTeX suitable form."
function compute-preconditions( this-sys : rf::system-analysis ) =
       let( *ANALYSIS-SC* : rlt::structure-chart =
              fret::extract-structure-chart-from-symbol-table(this-sys))
        let (program-units : seq(rf::program-unit) =
                             convert-analysis-into-subr-seq(this-sys))
           format(true, "\begindescription~%");
           (enumerate p over program-units do
              let ( pred-str : string = get-vars-from-subroutine(p))
                 if (~empty(pred-str))
                 then format(true, "\item[");
                      format(true,
                             symbol-to-string(
                                rf::identifier-name(
                                   rf::id-name(rf::unit-heading(p)))));
                      format(true, "] $");
                      format(true, pred-str);
                      format(true, "$~%"));
           format(true, "\enddescription~%")
"A specialization of compute-preconditions to the NAIF library"
function naif-preconditions () =
    let(system-name : string =
          "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis")
     let (this-sys : rf::system-analysis = analysis::load-system-analysis(
                                                          system-name))
       analysis::maybe-wrapup-system-analysis(this-sys);
       compute-preconditions(this-sys)
```

The file expr-to-latex.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  This file contains code for outputting expressions in the Fortran language
  model in a form suitable for processing by LaTeX.
||#
var *dummy-negate-for-test* : rf::negate-expression =
                              make-object('rf::negate-expression)
function literal-to-string( e : rf::expression ) : string =
  let(newstr  : string = [],
      the-lit : string =
                if (rf::literal-true(e))
                then " \bf true "
                elseif (rf::literal-false(e))
                then " \bf false "
                else format(false," ~\pp\ ", e))
%%
%% Go replicate all occurrences of ~ in strings.
%%
       (enumerate i over [1..length(the-lit)] do
          (if(the-lit(i) = #\~)
          then newstr <- append(newstr,#\~));
          newstr <- append(newstr, the-lit(i)));
       newstr
function id-to-string( id : rf::identifier ) : string =
  symbol-to-string(rf::identifier-name(id))
function expr-to-string( e : rf::expression ) : string
  computed-using
  rf::add-expression(e) => expr-to-string(e) = " + ",
  rf::and-expression(e) => expr-to-string(e) = " \wedge ",
  rf::concatenate-expression(e) => expr-to-string(e) = " . ",
  rf::divide-expression(e) => expr-to-string(e) = " / ",
  rf::eq-expression(e) => expr-to-string(e) = " = ",
  rf::eqv-expression(e) => expr-to-string(e) = " = ",
  rf::exponentiate-expression(e) => expr-to-string(e) = "^",
  rf::ge-expression(e) => expr-to-string(e) = " \geq ",
  rf::gt-expression(e) => expr-to-string(e) = " > ",
  rf::le-expression(e) => expr-to-string(e) = " \leq ",
```

```
  rf::lt-expression(e) => expr-to-string(e) = " < ",
  rf::multiply-expression(e) => expr-to-string(e) = " \ast ",
  rf::ne-expression(e) => expr-to-string(e) = " \neq ",
  rf::neqv-expression(e) => expr-to-string(e) = " \neq ",
  rf::or-expression(e) => expr-to-string(e) = " \vee ",
  rf::string-subrange-extraction(e) => expr-to-string(e) = " .. ",
  rf::subtract-expression(e) => expr-to-string(e) = " - ",
  rf::literal-constant(e) => expr-to-string(e) = literal-to-string(e),
  rf::open-key-parameter(e) => expr-to-string(e) = " ???? ",
  rf::special-intrinsic-call(e) => expr-to-string(e) = " ?? ",
  rf::identity-expression(e) => expr-to-string(e) = "",
  rf::negate-expression(e) => expr-to-string(e) = " - ",
  rf::not-expression(e) => expr-to-string(e) = " \neg "
function power-expr( e : rf::expression ) : boolean =
  rf::exponentiate-expression(e)
function multiplicative-expr( e : rf::expression ) : boolean =
  rf::divide-expression(e) or
  rf::concatenate-expression(e) or
  rf::string-subrange-extraction(e) or
  rf::multiply-expression(e)
function additive-expr( e : rf::expression ) : boolean =
  rf::add-expression(e) or
  rf::subtract-expression(e)
function atomic-expr( e : rf::expression ) : boolean =
  rf::identity-expression(e) or
  rf::literal-constant(e)
function eqv-expr(e : rf::expression ) : boolean =
  rf::eqv-expression(e) or
  rf::neqv-expression(e)
function eq-expr(e : rf::expression ) : boolean =
  rf::eq-expression(e) or
  rf::ne-expression(e) or
  rf::ge-expression(e) or
  rf::gt-expression(e) or
  rf::le-expression(e) or
  rf::lt-expression(e)
function and-or-expr( e : rf::expression ) : boolean =
  rf::and-expression(e) or
  rf::or-expression(e)
function not-expr( e : rf::expression ) : boolean =
```

```
  rf::negate-expression(e) or
  rf::not-expression(e)
function unknown-expr( e : rf::expression ) : boolean =
  rf::open-key-parameter(e) or
  rf::special-intrinsic-call(e)
#||
  Implements a total ordering over the types of expressions.
||#
function precedes-eq(e1 : rf::expression,
                     e2 : rf::expression ) : boolean =
  if ( atomic-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2) or
       multiplicative-expr(e2) or
       additive-expr(e2) or
       atomic-expr(e2)
  elseif ( additive-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2) or
       multiplicative-expr(e2) or
       additive-expr(e2)
  elseif ( multiplicative-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2) or
       multiplicative-expr(e2)
  elseif ( power-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2) or
       power-expr(e2)
```

```
  elseif ( eq-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2) or
       eq-expr(e2)
  elseif ( eqv-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2) or
       eqv-expr(e2)
  elseif ( and-or-expr(e1) )
  then not-expr(e2) or
       and-or-expr(e2)
  elseif ( not-expr(e1) )
  then not-expr(e2)
  else undefined
function need-parens?( e1 : rf::expression,
                       e2 : rf::expression ) : boolean =
  ~atomic-expr(e2) &
  ~rf::indexable-name(e2) &
  precedes-eq(e2,e1) &
  ~precedes-eq(e1,e2)
function construct-bin-expr-list( e : rf::expression,
                                  es : seq(rf::expression)) : string =
  if(empty(es))
  then undefined     %% This should never happen
  else let(head-xpr  : rf::expression = first(es),
           rest-xpr  : seq(rf::expression) = rest(es))
        let (head-p-str : string = expr-to-latex(head-xpr))
         let(head-str : string = if (need-parens?(e,head-xpr))
                                  then concat( "(", head-p-str, ")" )
                                 else head-p-str)
            if (~empty(rest-xpr))
            then let(rest-str : string = construct-bin-expr-list(e,rest-xpr))
                   (if power-expr(e)
                    then concat(head-str, expr-to-string(e), "",rest-str,"")
                    else concat(head-str, expr-to-string(e), rest-str))
            else head-str
function construct-binary-expr(e : rf::expression) : string =
  construct-bin-expr-list(e, rf::exp-list(e))
function construct-unary-expr(e : rf::expression) : string =
```

```
  if atomic-expr(e)
  then expr-to-string(e)
  else let( val   : rf::expression = rf::exp-value(e),
            x-str : string = expr-to-string(e))
          let ( val-str : string = expr-to-latex(val) )
             if (need-parens?(e,val))
             then concat(x-str,"(",val-str,")")
             else concat(x-str,val-str)
function construct-indexable-name(e : rf::expression) : string =
  let ( ret-str : string = id-to-string(rf::id-or-indexee(e)))
     (if (rf::indexable-name-has-params?(e))
      then let (params : seq(rf::expression-or-special) =
                         rf::function-params-or-indices(e))
             let (len : integer = length(params))
                if(len > 0) then
                  (ret-str <- concat(ret-str,"(",expr-to-latex(first(params)));
                   (enumerate i over [2..len] do
                       ret-str <- concat(ret-str, ",",
                                         expr-to-latex(params(i))));
                   ret-str <- concat(ret-str, ")")));
      ret-str
#||
  This function is grouped by binding precedence of the operators
  (to determine if parenthesis insertion is necessary).
||#
function expr-to-latex( e : rf::expression ) : string
  computed-using
  rf::indexable-name(e) => expr-to-latex(e) = construct-indexable-name(e),
  rf::binary-expression(e) => expr-to-latex(e) = construct-binary-expr(e),
  rf::literal-constant(e) => expr-to-latex(e) = literal-to-string(e),
  rf::open-key-parameter(e) => expr-to-latex(e) = "",
  rf::special-intrinsic-call(e) => expr-to-latex(e) = "",
  rf::unary-expression(e) => expr-to-latex(e) = construct-unary-expr(e)
function precond-to-latex( e       : rf::expression,
                           negate? : boolean ) : string =
  let (xpr-str : string = expr-to-latex(e))
     if (negate?)
     then (if (need-parens?(*dummy-negate-for-test*,e))
           then concat("\neg (", xpr-str, ")")
           else concat("\neg ", xpr-str))
```

```
    else xpr-str
```

7.2 Definite Loops

Definite loops are easily detected using Refine rule construct and a tree walking traversal.

The file def-loops.re

```
!! in-package("RU")
!! in-grammar('user)
var *number-of-do-loops* : integer = 0
var *number-of-definite-loops* : integer = 0
var *visited* : set(object) =
rule TALLY-STANDARD-DO-LOOP( a : object )
  ~(a in *visited*) &
  rf::do-statement(a) &
  rf::literal-constant(rf::base-value(a)) &
  rf::literal-constant(rf::exp-value(a))
    -->
    (*number-of-definite-loops* = *number-of-definite-loops* + 1) &
    (*number-of-do-loops* = *number-of-do-loops* + 1) &
    a in *visited*
rule COUNT-DO-LOOP( a : object )
  ~(a in *visited*) &
  rf::do-statement(a) &
  ~( rf::literal-constant(rf::base-value(a)) &
     rf::literal-constant(rf::exp-value(a)))
  -->
  (*number-of-do-loops* = *number-of-do-loops* + 1) &
  a in *visited*
var *LOOP-COUNT-RULES* : seq(symbol) =
  ['TALLY-STANDARD-DO-LOOP,
   'COUNT-DO-LOOP]
"Given a system-analysis, return the cumulative sequence of subroutines
 that are contained in this analysis."
function convert-analysis-into-subr-seq( this-system : rf::system-analysis ) :
  seq(rf::program-unit) =
  analysis::maybe-wrapup-system-analysis(this-system);
  let (exec-progs : seq(rf::executable-program) =
```

```
        [ e | (file-an : rf::file-analysis,
               e       : rf::executable-program)
               file-an in rf::file-analyses-for-system(this-system) &
               e = rf::file-analysis-file-ast(file-an) ])
    reduce(append, [ s | ( p : rf::executable-program,
                           s : seq(rf::program-unit) )
                         p in exec-progs &
                         s = rf::program-units(p) &
                         defined?(s) &
                         ~empty(s) ] )
"Given a system-analysis, return a tuple containing the number of subroutines
 in the system and the number of subroutines which contain either:
  1) no loops, or
  2) only loops with literal upper and lower bounds."
function compute-definite-loops( this-system : rf::system-analysis ) :
  tuple(integer,integer) =
  let (program-units : seq(rf::program-unit) =
                 convert-analysis-into-subr-seq(this-system),
       number-of-subrs      : integer = 0,
       number-of-unrollables : integer = 0)
  (enumerate ourguy over program-units do
   (let (*number-of-do-loops* : integer = 0,
         *number-of-definite-loops* : integer = 0,
         *visited* : set(object) = )
      preorder-transform(ourguy, *LOOP-COUNT-RULES*);
      number-of-subrs <- number-of-subrs + 1;
      *number-of-do-loops* = *number-of-definite-loops*
      -->
      number-of-unrollables = number-of-unrollables + 1));
   <number-of-unrollables, number-of-subrs>
function count-naif-loops() : tuple(integer,integer) =
  let(system-name : string =
          "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis")
    let (this-sys : rf::system-analysis = analysis::load-system-analysis(
                                                          system-name))
      let( dl-tup : tuple(integer,integer) =
                    compute-definite-loops(this-sys))
      format(true,
             "~% Found ~\pp\ subroutines of which ~\pp\",
             dl-tup.2,
```

```
        dl-tup.1);
    format(true,
        " had no or only definite loops~% for a ratio of ~\pp\~%",
        (integer-to-real(dl-tup.1) / integer-to-real(dl-tup.2)))
```

7.3 Detecting Constant Parameters

The structure chart object provided by Refine/Workbench contains information about subprograms and formal parameters. It also contains link to invocations of a particular procedure. To detect subprograms with constant parameters, simply enumerate over all subprograms and examine the sequence of call sites to see if a particular actual parameter is constant in every case.

The file const-pars.re

```
!! in-package("RU")
!! in-grammar('user)
#||
  $Id: const-pars.re,v 1.1 1995/12/19 19:05:03 kurt Exp $
  This code searches for routines in SPICELIB that are invoked somewhere
  else in the library with a "constant" parameter. Constant here means
  either a literal or a Fortran PARAMETER. The motivating hypothesis here
  is that invocations of routines with non-varying parameters may
  indicat control coupling.
  $Log: const-pars.re,v $
# Revision 1.1  1995/12/19  19:05:03  kurt
# Initial revision
#
||#
%%
%% We want to return true this call parameter is a constant
%% (literal or Fortran PARAMETER). To recognize this in the structure
%% chart is actually somewhat difficult. There are 2 cases:
%%    1) If a parameter is a literal, then its corresponding
%%       rf::call-argument must be undefined [indicating that it's
%%       not a variable] and an  empty set of rf::call-uses [indicating
%%       that it is not an expression].
%%
%%    2) If a parameter is a Fortran PARAMETER, then its corresponding
```

```
%%       rf::call-argument must have a "mode" rf::unit-mode of
%%       'FRET::CONSTANT [which I found out indicates that it's
%%       a PARAMETER].
%%
%%    3) If a parameter is an expression whose leaves are one of the
%%       above. rf::call-argument-descriptors represent only the
%%       leaves of the expressions in the set rf::call-uses.
%%
%%
function refs-only-constants?( r : rf::call-argument-descriptor ) : boolean =
  let (se : rf::symbol-entry = rf::call-argument(r))
    if ( se = undefined )
    then
       let (constant-refs : boolean = true)
          (enumerate i over rf::call-uses(r) do
             constant-refs <- (constant-refs & refs-only-constants?(i)));
          constant-refs
    else (rf::unit-mode(se) = 'FRET::CONSTANT)
function constant-params?( s : rf::call-descriptor ) : boolean =
  let (cargs : seq(rf::call-argument-descriptor) = rf::call-arguments(s))
     reduce(or,  refs-only-constants?(x) | (x) x in cargs )
"Returns a sequence of booleans, one for each invocation of the given node,
 that represent whether or not the node was invoked with constant parameters.
 The interpretation of this sequence is that its or reduction indicates
 that the node is called somewhere with constant parameters, and its and
 reduction indicates that it is called everywhere with constant parameters."
function calls-with-const-params( node : rlt::sc-node ) : seq(boolean) =
  let ( in-edges : set(rlt::sc-edge) = rlt::sc-node-in-edges(node) )
     [ constant-params?(y) | (x : rlt::sc-edge,
                              y : rf::program-unit-statement)
                             (x in in-edges) &&
                                  (y in rlt::sc-edge-call-points(x)) ]
"Gathers the set of routines that are invoked with constant parameters"
function gather-routines-called-constant-params (
        this-sys : rf::system-analysis ) :
                tuple(set(rlt::sc-node), set(rlt::sc-node)) =
  let (chart : rlt::structure-chart =
               fret::extract-structure-chart-from-symbol-table(this-sys))
    let (scnds           : set(rlt::sc-node) =
                   rlt::sc-nodes(chart),
```

```
        some-set : set(rlt::sc-node) = ,
        all-set : set(rlt::sc-node) =  )
      (enumerate i over scnds do
          if(rlt::sc-node-user-defined?(i)) then
             let(calls-with-consts : seq(boolean) =
                                      calls-with-const-params(i))
                (if(reduce(and, calls-with-consts)) then
                    (all-set <- all-set with i;
                     some-set <- some-set with i)
                 elseif(reduce(or, calls-with-consts)) then
                    some-set <- some-set with i));
     <all-set, some-set>
"The constant parameter to subprogram analysis specialized to the
 NAIF library"
function naif-const-param-analysis() =
  let (sys-name : string =
       "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis")
    let (this-sys : rf::system-analysis =
                    analysis::load-system-analysis(sys-name))
      analysis::maybe-wrapup-system-analysis(this-sys);
      let ( nodes : tuple(set(rlt::sc-node),set(rlt::sc-node)) =
            gather-routines-called-constant-params(this-sys) )
        format(true,"~% SUBPROGRAMS ALWAYS INVOKED WITH CONSTANT PARAMS~%");
        (enumerate i over nodes.1 do
            format(true, "~% ~\pp\~%", rlt::sc-node-name(i)));
        format(true,"~% SUBPROGRAMS INVOKED WITH CONSTANT PARAMS~%");
        (enumerate i over nodes.2 do
            format(true, "~% ~\pp\~%", rlt::sc-node-name(i)))
```

7.4 Multiple Outputs

The Software Refinery provides a package to create structure chart (call graph) objects. The nodes of these structure charts are annotated with direction information about parameters. We were thus able to detect subprograms with multiple outputs using only the structure chart object. This greatly simplified the implementation. The file `tuple.re` contains this analysis. Note that this detection returns the subprograms with multiple outputs that are invoked according to the domain theory.

```
!! in-package("RU")
!! in-grammar('user)
"This is the set of routines that we identified as directly reachable from
 the domain theory."
var *calls* : set(string) =
                                "BODMAT",
                                "BODVAR",
                                "CGV2EL",
                                "CKGPAV",
                                "CYLLAT",
                                "CYLREC",
                                "CYLSPH",
                                "EDLIMB",
                                "EL2CGV",
                                "EL2PL",
                                "ELPROJ",
                                "FINDPV",
                                "I2SC",
                                "INEDPL",
                                "INELPL",
                                "INRYPL",
                                "LATCYL",
                                "LATREC",
                                "MTXV",
                                "MXV",
                                "NEARPT",
                                "NPEDLN",
                                "NPELPT",
                                "NPLNPT",
                                "NVP2PL",
                                "PJELPL",
```

```
                                "PL2NVC",
                                "PSV2PL",
                                "RADREC",
                                "RECCYL",
                                "RECD",
                                "RECLAT",
                                "RECRAD",
                                "RECSPH",
                                "SCE2T",
                                "SENT",
                                "SPHCYL",
                                "SPHLAT",
                                "SPHREC",
                                "SPKEZ",
                                "SPKSSB",
                                "SRFREC",
                                "SURFNM",
                                "SURFPT",
                                "TIKTOL",
                                "VADD",
                                "VDIST",
                                "VEQU",
                                "VHAT",
                                "VMINUS",
                                "VPRJP",
                                "VSEP",
                                "VSUB"
function detect-mult-outs( cg : rlt::structure-chart ) : set(rlt::sc-node) =
   s | (s : rlt::sc-node)
         s in rlt::sc-nodes(cg) &
         ex(x : rlt::sc-formal,
            y : rlt::sc-formal)
           ( x in rlt::sc-node-formals(s) &
             y in rlt::sc-node-formals(s) &
             x ~= y &
             rlt::sc-formal-direction(x) in 'rlt::out,
                                            'rlt::in-out &
             rlt::sc-formal-direction(y) in 'rlt::out,
                                            'rlt::in-out)
function compute-mouts( this-sys : rf::system-analysis ) : set(string) =
```

```
#||
  let (this-sys : rf::system-analysis = analysis::load-system-analysis(
        "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis"))
||#
    analysis::maybe-wrapup-system-analysis(this-sys);
    let (cg : rlt::structure-chart =
              fret::extract-structure-chart-from-symbol-table(this-sys))
       let(multiple-outs : set(string) = image(rlt::sc-node-name,
                                               detect-mult-outs(cg)))
          intersect(multiple-outs, *calls*)
```

7.5 Amphion Domain Theory Coverage

Whenever a declarative system is used to generate code over a library, one would like to know whether or not the library is *covered* by the declarative domain. That is, can every routine in the library be invoked by *some* sentence in the domain theory, and is everything returned by a routine in the library accounted for in the domain theory? If the answer to either question is no, then either the library is superfluous or the domain theory is incomplete. We would like to know about either case.

Subprogram Reachability Detection

```
!! in-package("RU")
!! in-grammar('user)
%%
%% Want to conver the call graph (structure chart) into a relation so
%% that we can compute its transitive closure. Also, we want to remove
%% any edges to non user defined nodes.
%%
function call-graph-to-relation( cg : rlt::structure-chart ) :
  set(tuple(rlt::sc-node, rlt::sc-node)) =
  let(this-set : set(tuple(rlt::sc-node, rlt::sc-node)) = ,
      edges    : set(rlt::sc-edge) = rlt::sc-edges(cg))
     (e in edges &
      rlt::sc-node-user-defined?(rlt::sc-edge-to(e))
      -->
      <rlt::sc-edge-from(e), rlt::sc-edge-to(e)> in this-set);
     this-set
function to-string( s : set(tuple(rlt::sc-node, rlt::sc-node) ) ) :
  set(tuple(string, string)) =
   <rlt::sc-node-name(z.1), rlt::sc-node-name(z.2)> |
     (z : tuple(rlt::sc-node, rlt::sc-node)) z in s
"This is the set of routines that we identified as directly reachable from
 the domain theory."
var *calls* : set(string) =
                              "BODMAT",
                              "BODVAR",
                              "CGV2EL",
                              "CKGPAV",
                              "CYLLAT",
                              "CYLREC",
```

```
"CYLSPH",
"EDLIMB",
"EL2CGV",
"EL2PL",
"ELPROJ",
"FINDPV",
"I2SC",
"INEDPL",
"INELPL",
"INRYPL",
"LATCYL",
"LATREC",
"MTXV",
"MXV",
"NEARPT",
"NPEDLN",
"NPELPT",
"NPLNPT",
"NVP2PL",
"PJELPL",
"PL2NVC",
"PSV2PL",
"RADREC",
"RECCYL",
"RECD",
"RECLAT",
"RECRAD",
"RECSPH",
"SCE2T",
"SENT",
"SPHCYL",
"SPHLAT",
"SPHREC",
"SPKEZ",
"SPKSSB",
"SRFREC",
"SURFNM",
"SURFPT",
"TIKTOL",
"VADD",
```

```
                                "VDIST",
                                "VEQU",
                                "VHAT",
                                "VMINUS",
                                "VPRJP",
                                "VSEP",
                                "VSUB"
function get-results() : set(string) =
  let (this-sys : rf::system-analysis = analysis::load-system-analysis(
        "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis"))
       analysis::maybe-wrapup-system-analysis(this-sys);
       let (cg : rlt::structure-chart =
         fret::extract-structure-chart-from-symbol-table(this-sys))
         let ( closed-set : set(tuple(rlt::sc-node, rlt::sc-node)) =
              tclosure(call-graph-to-relation(cg)))
            let (closed-str-set : set(tuple(string, string)) =
                 to-string(closed-set))
            (enumerate i over image(closed-str-set, *calls*) do
              format(true,"~\pp\~%", i))
```

Dead End Dataflow Detection

```
!! in-package("RU")
!! in-grammar('user)
#||
  Given a set *COVERED-ROUTINES* of names of covered routines in the library,
  and given a set *COVERED-DATAFLOWS* of routine/parameter tuples, compute
  the dead-end dataflows in *COVERED-ROUTINES* with respect to
  *COVERED-DATAFLOWS*.
||#
var *empty-cp-desc* : covered-parameter-desc = <"", >
#||
  Should load this input from inputs.re (automatically generated by snark
                                         analysis).
var *COVERED-DATAFLOWS* : set(covered-parameter-desc) =
   <"LATCYL", "LONGC" >
||#
"Given a set of names of routines in the library and the call graph,
 return the set covered-parameter-descriptors that summarize these entities"
function expand-routine-names-from-cg( routine-names : set(string),
```

```
                                        cg          : rlt::structure-chart ) :
  set(covered-parameter-desc) =
  image( lambda (v : rlt::sc-node)
               < rlt::sc-node-name(v),
                  rlt::sc-formal-name(f) | (f : rlt::sc-formal)
                                           f in rlt::sc-node-formals(v) &
                                           rlt::sc-formal-direction(f) in
                                              'rlt::out, 'rlt::in-out  >,
         n | (n : rlt::sc-node) n in rlt::sc-nodes(cg) &
                                  rlt::sc-node-name(n) in routine-names )
"Display the pairs of routines with dataflows not covered."
function display-dead-end-dataflows(dead-ends : set(covered-parameter-desc)) =
  enumerate de over dead-ends do
     if( ~empty(de.out-params) ) then
        (format(true,
                "~\pp\ ",
                de.routine-name);
         (enumerate p over de.out-params do
             format(true,
                    "~\pp\ ", p));
         format(true, "~%"))
"Of the routines that do not seem to cover any outputs, go see if there
 are any outputs to cover."
function
compute-dead-end-dataflows-from-totals(covset : set(covered-parameter-desc))
  : set(covered-parameter-desc) =
   p | (p : covered-parameter-desc) p in covset & ~empty(p.out-params)
"Of the routines that cover some outputs, detect those outputs that are
 not covered."
function
compute-dead-end-dataflows-from-partials(covset : set(covered-parameter-desc))
  : set(covered-parameter-desc) =
  let( retset        : set(covered-parameter-desc) =  )
  %%
  %% Author's note. You might think a transform (-->) would be more
  %% appropriate here, and you would be right. Unfortunately, the
  %% transform equivalent of this unearthed a bug in Refine 4.0-Beta.
  %%
  fa(p : covered-parameter-desc)
     (p in covset =>
```

```
    fa(q : covered-parameter-desc)
       (q in *COVERED-DATAFLOWS* =>
          (p.routine-name = q.routine-name) =>
             retset <- retset with <p.routine-name,
                                    setdiff(p.out-params,q.out-params)>));
        retset
function analyze-dead-end-dataflows() =
  let (this-sys : rf::system-analysis = analysis::load-system-analysis(
        "/users/projects/groups/nasa-jpl/naif/analysis/naif-ast.analysis"))
    analysis::maybe-wrapup-system-analysis(this-sys);
    let (cg : rlt::structure-chart =
              fret::extract-structure-chart-from-symbol-table(this-sys),
         routine-names : set(string) =
                         image( lambda (x : covered-parameter-desc)
                                        x.routine-name,
                                *COVERED-DATAFLOWS* ))
      let( partial-routines : set(covered-parameter-desc) =
                              expand-routine-names-from-cg(routine-names, cg),
           total-routines   : set(covered-parameter-desc) =
                              expand-routine-names-from-cg(
                                  setdiff(*COVERED-ROUTINES*,routine-names),
                                  cg) )
        let(partials : set(covered-parameter-desc) =
                       compute-dead-end-dataflows-from-partials(
                                                    partial-routines),
            totals   : set(covered-parameter-desc) =
                       compute-dead-end-dataflows-from-totals(
                                                    total-routines))
            display-dead-end-dataflows(union(partials, totals))
```

Understanding Interleaved Code

Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{spencer, kurt, linda}@cc.gatech.edu

Abstract

Complex programs often contain multiple, interwoven strands of computation, each responsible for accomplishing a distinct goal. The individual strands responsible for each goal are typically delocalized and overlap rather than being composed in a simple linear sequence. We refer to these code fragments as being *interleaved*. Interleaving may be intentional–for example, in optimizing a program, a programmer might use some intermediate result for several purposes–or it may creep into a program unintentionally, due to patches, quick fixes, or other hasty maintenance practices. To understand this phenomenon, we have looked at a variety of instances of interleaving in actual programs and have distilled characteristic features. This paper presents our characterization of interleaving and the implications it has for tools that detect interleaving and extract the individual strands of computation. Our exploration of interleaving has been done in the context of a case study of a corpus of production mathematical software, written in Fortran from the Jet Propulsion Laboratory. This paper also describes our experiences in developing tools to detect interleaving in this software, driven by the application of program comprehension to improving the description of this software library's components. This in turn aids in the automated component-based synthesis of software using the library.

With every leaf a miracle.
— Walt Whitman.

1 Introduction

Imagine being handed a software system you have never seen before. Perhaps you need to track down a bug, rewrite the software in another language or extend it in some way. We know that software maintenance tasks such as these consume the majority of software costs [3], and we know that reading and understanding the code requires more effort than actually making the changes [9]. But we don't know what makes understanding the code itself so difficult.

Letovsky has observed that programmers engaged in software understanding activities typically ask "how" questions and "why" questions [17]. The former require an in-depth knowledge of the

programming language and the ways in which programmers express their software designs. This includes knowledge of common algorithms and data structures and even concerns style issues such as indentation and use of comments. Nevertheless, the answers to "how" questions can be derived from the program text. "Why" questions are more troubling. Answering them requires not only comprehending the program text but relating it to the program's *purpose*—solving some sort of problem. And the problem being solved may not be explicitly stated in the program text; nor is the rationale the programmer had for choosing the particular solution usually visible.

This paper is concerned with a specific difficulty that arises when trying to answer "why" questions about computer programs. In particular, it is concerned with the phenomenon of "interleaving" in which one section of a program accomplishes several purposes, and disentangling the code responsible for each purposes is difficult. Unraveling interleaved code involves discovering the purpose of each strand of computation, as well as understanding why the programmer decided to interleave the strands. To demonstrate this problem, we examine an example program in a step-by-step fashion, trying to answer the questions "why is this program the way it is?" and "what makes it difficult to understand?"

1.1 NPEDLN

The Fortran program, called **NPEDLN**, is part of the **SPICELIB** library obtained from the Jet Propulsion Laboratory and intended to help space scientists analyze data returned from space missions. The acronym **NPEDLN** stands for Nearest Point on Ellipsoid to Line. The ellipsoid is specified by the length of its three semi-axes (**A**, **B**, and **C**), which are oriented with the x, y, and z coordinate axes. The line is specified by a point (**LINEPT**) and a direction vector (**LINEDR**). The nearest point is contained in a variable called **PNEAR**. The full program consists of 565 lines; an abridged version can be found in the Appendix with a brief description of subroutines it calls and variables it uses. The executable statements, with comments and declarations removed, are shown in Figure 1.

The lines of code in **NPEDLN** that actually compute the nearest point are somewhat hard to locate. One reason for this has to with error checking. It turns out that **SPICELIB** includes an elaborate mechanism for reporting and recovering from errors, and roughly half of the code in **NPEDLN** is used for this purpose. We have indicated those lines by shading in Figure 2. The important point to note is that although it is natural to program in a way that intersperses error checks with computational code, it is not necessary to do so. In principal, an entirely separate routine could be constructed to make the checks and **NPEDLN** called only when all the checks are passed. Although this approach would require redundant computation and potentially more total lines of code, the resultant computations in **NPEDLN** would be shorter and easier to follow.

In some sense, the error handling code and the rest of the routine realize *independent* plans. We use the term *plan* to denote a description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program.[1] Plans can occur

[1] This definition is distilled from definitions in [18, 28, 34]. Note that a plan is not necessarily stereotypical or used repeatedly; it may be novel or idiosyncratic. Following [28, 34], we reserve the term *cliché* for a plan that represents a standard, stereotypical form, which can be detected by recognition techniques, such as [12, 17, 16, 25, 29, 40].

```
      SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
     .                    PNEAR, DIST )
C
      IF ( RETURN () ) THEN
         RETURN
      ELSE
         CALL CHKIN ( 'NPEDLN' )
      END IF
C
      CALL UNORM ( LINEDR, UDIR, MAG )
      IF ( MAG .EQ. 0 ) THEN
         CALL SETMSG( 'Line direction vector
     .                is the zero vector. ' )
         CALL SIGERR( 'SPICE(ZEROVECTOR)' )
         CALL CHKOUT( 'NPEDLN' )
         RETURN
      ELSE IF (          ( A .LE. 0.D0 )
     .         .OR.      ( B .LE. 0.D0 )
     .         .OR.      ( C .LE. 0.D0 ) )
     .   THEN
         CALL SETMSG ( 'Semi-axes: A = #,
     .                  B = #,  C = #.' )
         CALL ERRDP  ( '#', A )
         CALL ERRDP  ( '#', B )
         CALL ERRDP  ( '#', C )
         CALL SIGERR ('SPICE(INVALIDAXISLENGTH)')
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
      SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
      SCLA   =  A / SCALE
      SCLB   =  B / SCALE
      SCLC   =  C / SCALE
      IF (      ( SCLA**2  .LE.  0.D0 )
     .   .OR.   ( SCLB**2 .LE. 0.D0 )
     .   .OR.   ( SCLC**2 .LE. 0.D0 ) ) THEN
         CALL SETMSG ( 'Semi-axis too small:
     .                  A = #, B = #, C = #. ' )
         CALL ERRDP  ( '#', A )
         CALL ERRDP  ( '#', B )
         CALL ERRDP  ( '#', C )
         CALL SIGERR ('SPICE(DEGENERATECASE)')
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
C     Scale LINEPT.
      SCLPT(1)  =  LINEPT(1) / SCALE
      SCLPT(2)  =  LINEPT(2) / SCALE
      SCLPT(3)  =  LINEPT(3) / SCALE
      CALL VMINUS (UDIR, OPPDIR )
      CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
     .             SCLC, PT(1,1), FOUND(1))
      CALL SURFPT (SCLPT, OPPDIR, SCLA,
     .             SCLB, SCLC, PT(1,2), FOUND(2))

      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
            DIST  =  0.0D0
            CALL VEQU   ( PT(1,I),  PNEAR )
            CALL VSCL   ( SCALE,PNEAR, PNEAR )
            CALL CHKOUT ( 'NPEDLN' )
            RETURN
         END IF
50001 CONTINUE
C
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
     .              CAND, XFOUND )
      IF ( .NOT. XFOUND ) THEN
         CALL SETMSG ( 'Candidate ellipse could not
     .                  be found.' )
         CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C
      CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
     .              IFOUND )
      IF ( .NOT. IFOUND ) THEN
         CALL SETMSG ( 'Inverse projection could not
     .                  be found.' )
         CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
      CALL VSCL ( SCALE,  PNEAR,  PNEAR )
      DIST    =  SCALE * DIST
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
      END
```

Figure 1: **NPEDLN** minus comments and declarations.

```
      SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
     .                    PNEAR, DIST )
C
      IF ( RETURN () ) THEN
         RETURN
      ELSE
         CALL CHKIN ( 'NPEDLN' )
      END IF
C
      CALL UNORM ( LINEDR, UDIR, MAG )
      IF ( MAG .EQ. 0 ) THEN
         CALL SETMSG( 'Line direction vector
     .                is the zero vector. ' )
         CALL SIGERR( 'SPICE(ZEROVECTOR)' )
         CALL CHKOUT( 'NPEDLN' )
         RETURN
      ELSE IF (          ( A .LE. 0.D0 )
     .         .OR.      ( B .LE. 0.D0 )
     .         .OR.      ( C .LE. 0.D0 ) )
     .   THEN
         CALL SETMSG ( 'Semi-axes: A = #,
     .                  B = #,  C = #.' )
         CALL ERRDP  ( '#', A )
         CALL ERRDP  ( '#', B )
         CALL ERRDP  ( '#', C )
         CALL SIGERR ('SPICE(INVALIDAXISLENGTH)')
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
      SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
      SCLA   =  A / SCALE
      SCLB   =  B / SCALE
      SCLC   =  C / SCALE
      IF (     ( SCLA**2  .LE.  0.D0 )
     .  .OR. ( SCLB**2 .LE. 0.D0 )
     .  .OR. ( SCLC**2 .LE. 0.D0 ) )  THEN
         CALL SETMSG ( 'Semi-axis too small:
     .                  A = #, B = #, C = #. ' )
         CALL ERRDP  ( '#', A )
         CALL ERRDP  ( '#', B )
         CALL ERRDP  ( '#', C )
         CALL SIGERR ('SPICE(DEGENERATECASE)')
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
C     Scale LINEPT.
      SCLPT(1)  =  LINEPT(1) / SCALE
      SCLPT(2)  =  LINEPT(2) / SCALE
      SCLPT(3)  =  LINEPT(3) / SCALE
      CALL VMINUS (UDIR, OPPDIR )
      CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
     .             SCLC, PT(1,1), FOUND(1))
      CALL SURFPT (SCLPT, OPPDIR, SCLA,
     .             SCLB, SCLC, PT(1,2), FOUND(2))
C
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
            DIST  =  0.0D0
            CALL VEQU   ( PT(1,I),  PNEAR )
            CALL VSCL   ( SCALE,PNEAR, PNEAR )
            CALL CHKOUT ( 'NPEDLN' )
            RETURN
         END IF
50001 CONTINUE
C
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
     .              CAND, XFOUND )
      IF ( .NOT. XFOUND ) THEN
         CALL SETMSG ( 'Candidate ellipse could not
     .                  be found.'  )
         CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C
      CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
     .              IFOUND )
      IF ( .NOT. IFOUND ) THEN
         CALL SETMSG ( 'Inverse projection could not
     .                  be found.'  )
         CALL SIGERR ( 'SPICE(DEGENERATECASE)' )
         CALL CHKOUT ( 'NPEDLN' )
         RETURN
      END IF
      CALL VSCL ( SCALE,  PNEAR,  PNEAR )
      DIST    =   SCALE * DIST
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
      END
```

Figure 2: Code with error handling highlighted.

```
      SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
     .                    PNEAR, DIST )
C
      CALL UNORM ( LINEDR, UDIR, MAG )
      SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
      SCLA   =  A / SCALE
      SCLB   =  B / SCALE
      SCLC   =  C / SCALE
C
      SCLPT(1)  =  LINEPT(1) / SCALE
      SCLPT(2)  =  LINEPT(2) / SCALE
      SCLPT(3)  =  LINEPT(3) / SCALE
C
      CALL VMINUS (UDIR, OPPDIR )
      CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
     .             SCLC, PT(1,1), FOUND(1))
      CALL SURFPT (SCLPT, OPPDIR, SCLA,
     .             SCLB, SCLC, PT(1,2), FOUND(2))
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
            DIST  =  0.0D0
            CALL VEQU   ( PT(1,I),  PNEAR )
            CALL VSCL   ( SCALE,PNEAR, PNEAR )
            RETURN
         END IF
50001 CONTINUE
C
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
     .              CAND, XFOUND )
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C
      CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
     .              IFOUND )
      CALL VSCL ( SCALE,  PNEAR,  PNEAR )
      DIST    =   SCALE * DIST
      RETURN
      END
```

Figure 3: The residual code without the error handling plan.

at any level of abstraction from architectural overviews to code. By extracting the error checking plan from **NPEDLN**, we get the much smaller and, presumably, more understandable program shown in Figure 3.

The structure of an understanding process begins to emerge: detect a plan, such as error checking, in the code and extract it, leaving a smaller and more coherent residue for further analysis; document the extracted plan independently; and note the way in which it interacts with the rest of the code.

We can apply this approach further to **NPEDLN**'s residual code in Figure 3. **NPEDLN** has a primary goal of computing the nearest point on an ellipsoid to a specified line. It also has an orthogonal goal of ensuring that the computations involved have stable numerical behavior; that is, that the computations are accurate in the presence of a wide range of numerical inputs. A standard trick in numerical programming for achieving stability is to scale the data involved in a computation and then unscale the results. The code responsible for doing this in **NPEDLN** is scattered throughout the program's text. It is highlighted in the excerpt shown in Figure 4.

The *delocalized* nature of this "scale-unscale" plan makes it difficult to gather together all the pieces involved for consistent maintenance. It also gets in the way of understanding the rest of the code, since it provides distractions that must be filtered out. Letovsky and Soloway's cognitive study [18] shows the deleterious effects of delocalization on comprehension and maintenance.

When we extract the scale-unscale code from **NPEDLN**, we are left with a much smaller code segment shown in Figure 5 that more directly expresses the program's purpose: computing the nearest point.

```
      SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
     .                    PNEAR, DIST )
C
      CALL UNORM ( LINEDR, UDIR, MAG )
      SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
      SCLA   =  A / SCALE
      SCLB   =  B / SCALE
      SCLC   =  C / SCALE
C
      SCLPT(1)  =  LINEPT(1) / SCALE
      SCLPT(2)  =  LINEPT(2) / SCALE
      SCLPT(3)  =  LINEPT(3) / SCALE
C
      CALL VMINUS (UDIR, OPPDIR )
      CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
     .             SCLC, PT(1,1), FOUND(1))
      CALL SURFPT (SCLPT, OPPDIR, SCLA,
     .             SCLB, SCLC, PT(1,2), FOUND(2))
      DO 50001
     .    I = 1, 2
        IF ( FOUND(I) ) THEN
           DIST  =  0.0D0
           CALL VEQU   ( PT(1,I),  PNEAR )
           CALL VSCL   ( SCALE,PNEAR, PNEAR )
           RETURN
        END IF
50001 CONTINUE
C
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
     .              CAND, XFOUND )
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C
      CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
     .              IFOUND )
      CALL VSCL ( SCALE,  PNEAR,  PNEAR )
      DIST   =  SCALE * DIST
      RETURN
      END
```

Figure 4: Code with scale-unscale plan highlighted.

```
      SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
     .                    PNEAR, DIST )
C
      CALL UNORM ( LINEDR, UDIR, MAG )
C
      CALL VMINUS (UDIR, OPPDIR )
      CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
     .             SCLC, PT(1,1), FOUND(1))
      CALL SURFPT (SCLPT, OPPDIR, SCLA,
     .             SCLB, SCLC, PT(1,2), FOUND(2))
      DO 50001
     .    I = 1, 2
        IF ( FOUND(I) ) THEN
           DIST  =  0.0D0
           CALL VEQU   ( PT(1,I),  PNEAR )
           RETURN
        END IF
50001 CONTINUE
C
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
     .              CAND, XFOUND )
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C
      CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
     .              IFOUND )
      RETURN
      END
```

Figure 5: The residual code without the scale-unscale plan.

```
      SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
     .                    PNEAR, DIST )
C
      CALL UNORM ( LINEDR, UDIR, MAG )
C
      CALL VMINUS (UDIR, OPPDIR )
      CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
     .             SCLC, PT(1,1), FOUND(1))
      CALL SURFPT (SCLPT, OPPDIR, SCLA,
     .             SCLB, SCLC, PT(1,2), FOUND(2))
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
            DIST = 0.0D0
            CALL VEQU  ( PT(1,I),  PNEAR )
            RETURN
         END IF
50001 CONTINUE
C
      NORMAL(1) = UDIR(1) / SCLA**2
      NORMAL(2) = UDIR(2) / SCLB**2
      NORMAL(3) = UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
     .              CAND, XFOUND )
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C
      CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
     .              IFOUND )
      RETURN
      END
```

Figure 6: Code with distance plan highlighted.

There is one further complication, however. It turns out that **NPEDLN** not only computes the nearest point from a line to an ellipsoid, it also computes the shortest distance between the line and the ellipsoid. This additional output (**DIST**) is convenient to construct, based on intermediate results obtained while computing the primary output (**PNEAR**). This is illustrated in Figure 6.[2]

Note that an alternative way to structure **SPICELIB** would be to have *independent* routines for computing the nearest point and the distance. The two routines would each be more coherent, but the common intermediate computations would have to be repeated, both in the code and at runtime.

The "pure" nearest point computation is shown in Figure 7. It is now much easier to see the primary computational purpose of this code.

The production version of **NPEDLN** contains several interleaved plans. Intermediate Fortran computations are shared by the nearest point and distance plans. A delocalized scaling plan is used to improve numerical stability, and an independent error handling plan is used to deal with unacceptable input. Knowledge of the existence of the several plans, how they are related and why they were interleaved is required for a deep understanding of **NPEDLN**.

1.2 Contributions

In this paper, we present a characterization of interleaving incorporating three aspects that make interleaved code difficult to understand: independence, delocalization, and resource sharing. We have distilled this characterization from an empirical examination of existing software—primarily

[2] The computation of **DIST** using **VDIST** is actually the last computation performed by the subroutine **NPELPT**, which **NPEDLN** calls; we have pulled this computation out of **NPELPT** for clarity of presentation.

```
      SUBROUTINE NPEDLN ( A, B, C, LINEPT, LINEDR,
     .                    PNEAR, DIST )
C
      CALL UNORM ( LINEDR, UDIR, MAG )
C
      CALL VMINUS (UDIR, OPPDIR )
      CALL SURFPT (SCLPT, UDIR, SCLA, SCLB,
     .             SCLC, PT(1,1), FOUND(1))
      CALL SURFPT (SCLPT, OPPDIR, SCLA,
     .             SCLB, SCLC, PT(1,2), FOUND(2))
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
            CALL VEQU   ( PT(1,I),  PNEAR )
            RETURN
         END IF
50001 CONTINUE
C
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL ( SCLA, SCLB, SCLC, CANDPL,
     .              CAND, XFOUND )
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT  )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
C
      CALL VPRJPI ( PRJNPT, PRJPL, CANDPL, PNEAR,
     .              IFOUND )
      RETURN
      END
```

Figure 7: The residual code without the distance plan.

SPICELIB. Secondary sources of existing software which we also examined are a Cobol database report writing system from the US Army and a program for finding the roots of functions (ZEROIN), presented and analyzed in [1] and [30]. We relate our characterization of interleaving to existing concepts in the literature, such as delocalized plans [18], coupling [41], and redistribution of intermediate results [10, 11].

We then describe the context in which we are exploring and applying these ideas. Our driving program comprehension problem is to elaborate and validate existing partial specifications of the JPL library routines to facilitate the automation of specification-driven generation of programs using these routines. We are developing analysis tools, based on the Software Refinery, to detect interleaving. We describe the analyses that we have formulated to detect classes of interleaving that are particularly useful to elaborating specifications. This paper concludes with a reflection on the feasibility of building tools to assist interleaving detection and extraction, open issues in the requirements on software and plan representations that detection imposes, the role of domain model guidance and cliché recognition in addressing the interleaving problem, and issues that arise when scaling up these analyses to the architectural level.

2 Interleaving

Programmers solve problems by breaking them into pieces. Pieces are programming language implementations of plans, and it is common for multiple plans to occur in a single code segment. We use the term "interleaving" to denote this merging.

> *Interleaving* expresses the merging of two or more distinct plans within some contiguous textual area of a program. Interleaving can be characterized by the *delocalization* of the code for the individual plans involved, the *sharing* of some resource, and the implementation of multiple, *independent* plans in the program's overall purpose.

We are characterizing the types of interleaving that typically occur in programs in order to develop techniques for detecting and extracting interleaved, but logically cohesive plans.

Interleaving may arise for several reasons. It may be intentionally introduced for efficiency. For example, it may be more efficient to compute two related values in one place than to do so separately. Intentional interleaving may also be performed to deal with non-functional requirements, such as numerical stability, that impose global constraints which are satisfied by diffuse computational structures. Interleaving may also creep into a program unintentionally, as a result of inadequate software maintenance, such as adding a feature locally to an existing routine rather than undertaking a thorough redesign. Or interleaving may arise as a natural by-product of expressing separate but related plans in a linear, textual medium. For example, accessors and constructors for manipulating data structures are typically interleaved throughout programs written in traditional programming languages due to their procedural, rather than object-oriented structure. Regardless of why interleaving is introduced, it severely complicates understanding a program. This makes it difficult to perform tasks such as extracting reusable components, localizing the effects of maintenance changes, and migrating to object-oriented languages.

There are several reasons interleaving is a source of difficulties. The first has to do with delocalization. Because two or more design purposes are implemented in a single segment of code, the individual code fragments responsible for each purpose are more spread out than they would be if they were encapsulated. Another reason interleaving presents a problem is that when it is the result of poorly thought out maintenance activities, the original, highly coherent structure of the system is typically degraded as "patches" and "quick fixes" are introduced. Finally, while interleaving may be introduced for purposes of optimization, expressing intricate optimizations in a clean and well-documented fashion is not typically done. The rationale behind the decision to intentionally introduce interleaving is often not explicitly recorded in the program. For all of these reasons, our ability to comprehend code containing interleaved fragments is compromised.

We now examine each of the characteristics of interleaving—delocalization, sharing, independence—in more detail.

2.1 Delocalization

Delocalization is one of the key characteristics of interleaving: one or more parts of a plan are spatially separated from other parts by code from other plans with which they are interleaved.

The "scale-unscale" pattern found in `NPEDLN` is a simple example of a more general delocalized plan that we refer to as a *reformulation wrapper*, which is frequently interleaved with computations in `SPICELIB`. Reformulation wrappers transform one problem into another that is simpler to solve

```
SUBROUTINE NPEDLN(A, B, C, LINEPT, LINEDR,
.                 PNEAR, DIST)
...
CALL UNORM ( LINEDR, UDIR, MAG )
... [error checks]
SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
SCLA   =  A / SCALE
SCLB   =  B / SCALE
SCLC   =  C / SCALE
... [error checks]
SCLPT(1)  =  LINEPT(1) / SCALE
SCLPT(2)  =  LINEPT(2) / SCALE
SCLPT(3)  =  LINEPT(3) / SCALE
CALL VMINUS ( UDIR, OPPDIR )
CALL SURFPT ( SCLPT, UDIR, SCLA, SCLB,
.             SCLC,PT(1,1), FOUND(1))
CALL SURFPT ( SCLPT, OPPDIR,SCLA, SCLB,
.             SCLC, PT(1,2), FOUND(2))
... [checking for intersection of the
     line with the ellipsoid]
    IF ( FOUND(I) ) THEN
       DIST  =  0.0D0
       CALL VSCL   (SCALE, PNEAR, PNEAR)
       ...
       RETURN
    END IF
... [handling the non-intercept case]
CALL VSCL ( SCALE,  PNEAR,  PNEAR )
DIST   =   SCALE * DIST
...
RETURN
END
```

Figure 8: Portions of the **NPEDLN** Fortran program. Shaded regions highlight the lines of code responsible for scaling and unscaling.

and then transfer the solution back to the original situation. Other examples of reformulation wrappers in **SPICELIB** are reducing a three-dimensional geometry problem to a two-dimensional one and mapping an ellipsoid to the unit sphere to make it easier to solve intersection problems.

Delocalization may occur for a variety of reasons. One is that there may be an inherently non-local relationship between the components of the plan, as is the case with reformulation wrappers, which makes the spatial separation necessary. Another reason is that the intermediate results of part of a plan may be shared with another plan, causing the plans to overlap and their steps to be shuffled together; the steps of one plan separate those of the other. For example, in Figure 8, part of the scale plan (computing the scaling factor) is separated from the rest of the plan (dividing by the scaling factor) in all scalings, except the scaling of **A**. This allows the scaling factor to be computed once and the result reused.

Realizing that a reformulation wrapper or some other delocalized plan is interleaved with a particular computation can help prevent severe comprehension failures during maintenance [18]. It can also help detect when the delocalized plan is incomplete, as it was in an earlier version of our example subroutine whose modification history includes the following correction:

```
C- SPICELIB Version 1.2.0, 25-NOV-1992 (NJB)
C Bug fix: in the intercept case, PNEAR is now
C properly re-scaled prior to output.  Formerly,
C it was returned without having been re-scaled.
```

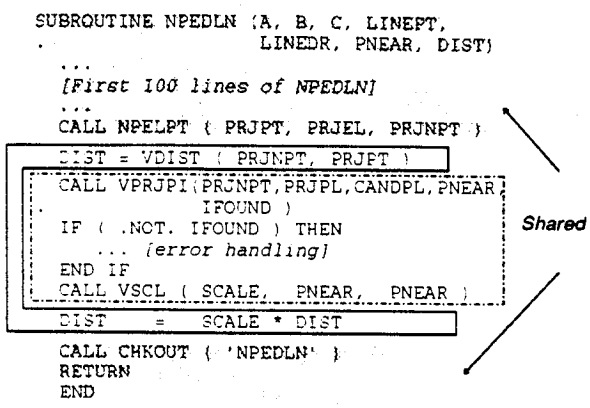

Figure 9: Portions of **NPEDLN**, highlighting two overlapping computations.

2.2 Resource Sharing

The sharing of some resource is characteristic of intentional interleaving. When interleaving is introduced into a program, there is normally some implicit relationship between the interleaved plans, motivating the designer to choose to interleave them. An example of this within **NPEDLN** is shown in Figure 9. The shaded portions of the code shown are *shared* between the two computations for **PNEAR** and **DIST**. In this case, the common resources shared by the interleaved plans are intermediate data results. The implementations for computing the nearest point and the shortest distance overlap in that a single structural element contributes to multiple goals.

The sharing of the results of some subcomputation in the implementation of two distinct higher level operations is termed *redistribution of intermediate results* by Hall [10, 11]. More specifically, redistribution is a class of function sharing optimizations which are implemented simply by tapping into the dataflow from some value producer and feeding it to an additional target consumer, introducing fanout into the dataflow. Redistribution covers a wide range of common types of function sharing optimizations, including common subexpression elimination and generalized loop fusion. Hall developed an automated technique for redistributing results for use in optimizing code generated from general-purpose reusable software components. We are interested in "undoing" these types of optimizations, and we can use the redistribution concept to capture forms of interleaving in which the resources shared are *data* values.

The commonality between interleaved plans might be in the form of other shared resources besides data values, such as control structures, lexical module structures, and names.

Control coupling. Control conditions may be redistributed just as data values are. The use of control flags allows control conditions to be determined once but used to affect execution at more than one location in the program. In **NPEDLN**, for example, **SURFPT** is called to compute the intersection of the line with the ellipsoid. This routine returns a control flag, **FOUND**, indicating whether or not the intersection exists. This flag is then used outside of **SURFPT** to control whether the intercept or

```
      CALL SURFPT ( SCLPT, UDIR,   SCLA, SCLB,
     .              SCLC, PT(1,1), FOUND(1) )
      CALL SURFPT ( SCLPT, OPPDIR, SCLA, SCLB,
     .              SCLC, PT(1,2), FOUND(2) )
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
           ... [handling the intercept case]
           RETURN
         END IF
50001 CONTINUE
C     Getting here means the line doesn't
C     intersect the ellipsoid.
      ... [handling the non-intercept case]
      RETURN
      END
```

Figure 10: Fragment of subprogram showing control coupling

non-intercept case is to be handled, as is shown in Figure 10.

The use of control flags is a special form of *control coupling*: "any connection between two modules that communicates elements of control [41]," typically in the form of function codes, flags, or switches [21]. This sharing of control information between two modules increases the complexity of the code, complicating comprehension and maintenance.

Content coupling. Another form of resource sharing occurs when the lexical structure of a module is shared among several related functional components. For example, the entire contents of a module may be lexically included in another. This sometimes occurs when a programmer wants to take advantage of a powerful intraprocedural optimizer limited to improving the code in a single routine. Another example occurs when a programmer uses `ENTRY` statements to partially overlap the contents of several routines so that they may share access to some state variables. This is sometimes done in a language, such as Fortran, that does not contain an encapsulation mechanism like packages or objects.

These two practices are examples of a phenomenon called *content coupling* in which [41] — "some or all of the contents of one module are included in the contents of another"—and which often manifests itself in the form of a multiple-entry module. Content coupling makes it difficult to independently modify or maintain the individual functions.

Name Sharing. A simple form of sharing is the use of the same variable name for two different purposes. This can lead to incorrect assumptions about the relationship between subcomputations within a program.

In general, the difficulty that resource sharing introduces is that it causes ambiguity in interpreting the purpose of program pieces. This can lead to incorrect assumptions about what effect changes will have, since the maintainer might be focusing on only one of the actual uses of the

resource (variable, value, control flag, data structure slot, etc.).

2.3 Independence

While interleaving is introduced to take advantage of commonalities, it is also true that the interleaved plans each have a distinct purpose. One implication of this is that the decision to interleave the plans can, in principle, always be undone. This may require copying of common code to eliminate resource sharing, resulting in an equivalent, but possibly less efficient, program.

In the NPEDLN example, a separate routine could be provided responsible for computing DIST. This code would be nearly identical to the original, requiring added maintenance and a likely run-time cost as well. Although interleaving may be necessary for efficiency, it obscures the independence of the components involved. Ironically, this hinders activities for making the code more efficient and reusable in the long run, such as parallelization and objectivization of the code.

3 Case Study

In order to better understand interleaving, we have undertaken a case study of production library software. The library, called SPICELIB, consists of approximately 600 mathematical programs, written in Fortran by programmers at the Jet Propulsion Laboratory for analyzing data sent back from space missions. The software performs calculations in the domain of solar system geometry, such as coordinate frame conversions, intersections of rays, ellipses, planes, and ellipsoids, and light-time calculations. NPEDLN comes from this library.

We were introduced to SPICELIB by researchers at NASA Ames, who have developed a component-based software synthesis system, called Amphion [20, 19, 37]. Amphion automatically constructs programs that compose routines drawn from SPICELIB. It does this by making use of a *domain theory* that includes formal specifications of the library routines, connecting them to abstract concepts in the solar system geometry domain. The knowledge of the domain is encoded in a structured representation, expressed as axioms in first-order logic with equality. A space scientist using Amphion can schematically specify the geometry of a problem through a graphical user interface, and Amphion automatically generates Fortran programs to call SPICELIB routines to solve the described problem. Amphion is able to do this by proving a theorem about the solvability of the problem in the domain and, as a side effect, generating the appropriate calls. This is shown in the bottom half of Figure 11.

Amphion has been installed at JPL and used by space scientists to successfully generate over one hundred programs to solve solar system kinematics problems. The programs consist of dozens of subroutine calls and are typically synthesized in under three minutes of CPU time using a Sun Sparc 2 [20, 19]. Amphion's success depends on how accurate, consistent, and complete the domain theory is that Amphion uses. An essential program understanding task is to validate the domain theory by checking it against the SPICELIB routines and extending it when incompletenesses are found.

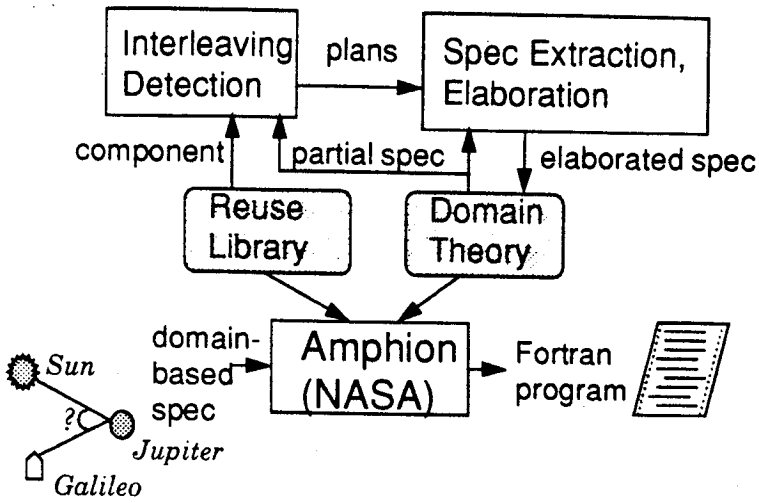

Figure 11: Applying interleaving detection to component-based reuse.

To do this, we need to be able to pull apart interleaved strands. For example, one incompleteness in Amphion's domain theory is that it does not fully cover the functionality of the routines. Some routines compute more than one result. For example, **NPEDLN** computes the nearest point on an ellipsoid to a line as well as the shortest distance between that point and the ellipsoid. However, the domain theory does not always describe all the values that are computed. In the case of **NPEDLN**, only the nearest point computation is modelled, not the shortest distance. In these routines, it is often the case that the code responsible for the secondary functionalities is interleaved with the code for the primary function covered by Amphion's domain theory. Uncovering the secondary functionality requires unraveling and understanding two interleaved computations.

Another way in which Amphion's current domain theory is incomplete is that it does not express preconditions on the use of the library routines; for example, when a line given as input to a routine is not the zero vector or that an ellipsoid's semi-axes must be large enough to be scalable. It is difficult to detect the code responsible for checking these preconditions because it is usually tightly interleaved with the code for the primary computation in order to take advantage of intermediate results computed for the primary computation.

In collaboration with NASA Ames researchers, we are detecting ways in which Amphion's domain theory is incomplete, and we are building program comprehension techniques to extend it. As the top half of Figure 11 shows, we are developing mechanisms for detecting particular classes of interleaving, with the aim of extending a partial model of the software's application domain. In the process, we are also performing analyses to gather empirical information about how much of **SPICELIB** is covered by the domain theory.

We are building interleaving detection mechanisms and empirical analysis tools using a collection of commercial tools, called the Software Refinery [14]. This is a comprehensive tool suite including

language-specific analyzers and browsers for Fortran, C, Ada, and Cobol, language extension mechanisms for building new analyzers, and a user interface construction tool for displaying the results of analysis. It maintains an object-oriented repository for holding the results of analyses, such as abstract syntax trees and symbol tables. It provides a powerful wide-spectrum language, called Refine [35], which supports pattern matching and querying the repository. Using the Software Refinery allows us to leverage commercially available tools as well as evaluate the strengths and limitations of its approach to program analysis, which we discuss in Section 4.1.

Section 3.1 briefly describes our motivation for building tools to elaborate a domain model, from the perspective of both synthesis and analysis. Section 3.2 describes mechanisms for detecting and reporting precondition checks in the library routines. Section 3.3 describes heuristic mechanisms still under development for finding candidate places where interleaving is likely to occur.

3.1 Domain Model Elaboration in Synthesis and Analysis

Our motivations for validating and extending a partial domain model of existing software come both from the synthesis and from the analysis perspectives. The primary motivations for doing this from the *synthesis* perspective are to make component retrieval more accurate, to assist in updating and growing the domain model as new software components are added, and to improve the software synthesized.

From the software *analysis* perspective, the refinement and elaboration of domain knowledge, based on what is discovered in the code, is a primary activity, driving the generation of hypotheses and informing future analyses. The process of understanding software involves two parallel knowledge acquisition activities [5, 23, 36]:

1. using domain knowledge to understand the code—knowledge about the application sets up expectations about how abstract concepts are typically manifested in concrete code implementations;

2. using knowledge of the code to understand the domain—what is discovered in the code is used to build up a description of various aspects of the application and to help answer questions about why certain code structures exist and what is their purpose with respect to the application.

We are studying interleaving in the context of performing these activities, given `SPICELIB` and an incomplete model of its application domain. We are targeting our detection of interleaving toward elaborating the existing domain model. We are also looking for ways in which the current knowledge in the domain model can guide detection and ultimately comprehension.

```
C$Procedure      SURFPT ( Surface point on an ellipsoid )
      SUBROUTINE SURFPT ( POSITN, U, A, B, C, POINT, FOUND )
      DOUBLE PRECISION U      ( 3 )
      ...declarations...
C     Check the input vector to see if its the zero vector. If it is
C     signal an error and return.
C
      IF  ( ( U(1) .EQ. 0.0D0 ) .AND.
     .      ( U(2) .EQ. 0.0D0 ) .AND.
     .      ( U(3) .EQ. 0.0D0 )        ) THEN
         CALL SETMSG ( 'SURFPT: The input vector is the zero vector.' )
         CALL SIGERR ( 'SPICE(ZEROVECTOR)' )
         CALL CHKOUT ( 'SURFPT' )
         RETURN
      END IF
      ...
```

Figure 12: A fragment of the subroutine SURFPT in SPICELIB. This fragment shows a precondition check which invokes an exception if all of the elements of the U array are 0.

3.2 Extracting Preconditions

Using the Software Refinery, we automated a number of program analyses, one of which is the detection of subroutine parameter precondition checks. Because precondition checks are often interspersed throughout a subprogram, they tend to delocalize the plans that perform the primary computational work. They are usually part of a larger plan that detects exceptional, usually erroneous conditions in the state of a running program, and then takes alternative action when these conditions arise, such as returning with an error code, signaling, or invoking error handlers.

We found many examples of precondition checks in our empirical analysis of the SPICELIB. One such check occurs in the subprogram SURFPT and is shown in Figure 12. SURFPT finds the intersection (POINT) of a ray (represented by a point POSITN and a direction vector U) with an ellipsoid (represented as three semi-axes lengths A, B, and C), if such an intersection exists (indicated by FOUND). One of the preconditions checked by SURFPT is that the direction vector U is not the zero-vector.

Precondition checks make explicit the assumptions a subprogram places on its inputs. The process of understanding a subprogram can be facilitated by removing its precondition checks and using the information they encode to elaborate a high-level specification of the subprogram. We have created a tool that detects precondition checks and extracts the preconditions into a documentation form suitable for expression as a partial specification. The specifications can then be compared against the Amphion domain model.

Precondition checks are particularly difficult to understand when they are sprinkled throughout the code of a subroutine as opposed to being localized at the beginning. We discovered that, though interleaved, these checks could be reliably identified by searching for IF statements whose conditions are a function of the input parameters and whose bodies invoke exception handlers. The analysis

RECGEO $\neg(F \geq 1) \wedge \neg(RE \leq 0.0D0)$

REMSUB $\neg((LEFT > RIGHT) \vee (RIGHT < 1) \vee (LEFT < 1) \vee (RIGHT > LEN(IN)) \vee (LEFT > LEN(IN)))$

SURFPT $\neg((U(1) = 0.0D0) \wedge (U(2) = 0.0D0) \wedge (U(3) = 0.0D0))$

XPOSBL $\neg((MOD(NCOL, BSIZE) \neq 0) \vee (MOD(NROW, BSIZE) \neq 0)) \wedge \neg(NCOL < 1) \wedge \neg(NROW < 1) \wedge \neg(BSIZE < 1)$

Figure 13: Preconditions extracted for some of the subroutines in SPICELIB.

variables through the sequence of statements in the subroutine. At each statement, if a variable X in the set could be modified by the execution of the statement, then X is removed from the set. After the propagation, we can easily check whether or not an IF statement is a guard.

Results The result of this analysis is a table of preconditions associated with each subroutine. Since we are targetting partial specification elaboration, we chose to make the tool output the preconditions in LaTeX form so as to generate nicely formatted reports. Figure 13 gives examples of preconditions extracted for a few SPICELIB subroutines. Our tool generated the LaTeX source included in Figure 13 without change.

Taken literally, the precondition for SURFPT, for example, states that one of the first three elements of the U array parameter must be non-zero. In the domain model, U is seen as a vector, so the more abstract precondition can be stated as "U is not the zero vector." Extracting the precondition into the literal representation is the first step to being able to express the precondition in the more abstract form.

The other preconditions listed in Figure 13, stated in their abstract form, are the following. The subroutine RECGEO converts the rectangular coordinates of a point RECTAN to geodetic coordinates, with respect to a given reference spheroid whose equatorial radius is RE, using a flattening coefficient F. Its precondition is that the radius is greater than 0 and the flattening coefficient is less than 1. The subroutine REMSUB removes the substring (LEFT:RIGHT) from a character string IN. It requires that the positions of the first character LEFT and the last character RIGHT to be removed are in the range 1 to the length of the string and that the position of the first character is less than the position of the last. Finally, the subroutine XPOSBL transposes the square blocks within a matrix BMAT. Its preconditions are that the block size BSIZE must evenly divide both the number of rows NROW in BMAT and the number of columns NCOL and that the block size, number of rows, and number of columns are all at least 1.

3.3 Finding Interleaving Candidates

There are other analyses that we are implementing which are heuristic techniques for finding likely interleaving candidates.

3.3.1 Routines with Multiple Outputs

One heuristic for finding instances of interleaving is to determine which subroutines compute more than one output. When this occurs, the subroutine is returning either the results of multiple distinct computations or a result whose type can not be directly expressed in the Fortran type system (e.g., as a data aggregate). In the former case, the subroutine is realized as the interleaving of multiple distinct plans, as is the case with NPEDLN's computation of both the nearest point and the shortest distance.

In the latter case, the subroutine may be implementing only a single plan, but a maintainer's conceptual categorization of the subroutine is still obscured by the appearance of some number of seemingly distinct outputs. A good example of this case occurs in the SPICELIB subroutine SURFPT, which conceptually returns the intersection of a vector with the surface of an ellipsoid. However, it is possible to give SURFPT a vector and an ellipsoid that do not intersect. In such a situation the output parameter POINT will be undefined, but the Fortran type system cannot express the type: DOUBLE PRECISION $\vee$ *Undefined*. The programmer was forced to simulate a variable of this type using two variables, POINT and FOUND, adopting the convention that when FOUND is **false**, the return value is *Undefined*, and when FOUND is **true**, the return value is POINT.

Clearly subprograms with multiple outputs complicate program understanding. We built a tool that determines the multiple output subprograms in a library by analyzing the direction of dataflow in parameters of functions and subroutines. A parameter's direction is either: **in** if the parameter is only read in the subprogram, **out** if the parameter is only written in the subprogram, or **in-out** if the parameter is both read and written in the subprogram. Multiple output subprograms will have more than one parameter with direction out or in-out.

Our tool bases its analysis on the structure chart (call graph) objects that the Software Refinery creates. The nodes of these structure charts are annotated with direction information about parameters. The resulting analysis showed that 25 percent of the subprograms in SPICELIB had multiple output parameters. We were thus able to focus our work on these routines first, as they are likely to involve interleaving.

In addition, we performed an empirical analysis to determine, for those routines covered by the Amphion domain model (35 percent of the library), which ones have multiple output parameters, some of which are not covered by the domain model. We refer to outputs that are not mapped to anything in the domain model as *dead end dataflows*, since the programs that Amphion creates can never make use of these return values; they have not been associated with any meaning in the application domain. For example, NPEDLN's distance output (DIST) is a dead end dataflow as far as the domain model is concerned. Dead end dataflows imply interleaving in the subprogram and/or an incompleteness in the domain model. Our analysis revealed that of the subroutines covered by the domain model, 30 percent have some output parameters that are dead end dataflows. These are good focal points for detecting interleaved plans that might be relevant to elaborating the domain theory.

3.3.2 Control Coupling

Another heuristic for detecting potential interleaving finds candidate routines that may be involved in *control coupling*. In particular, it asks the question: *Which calls to library routines, supply a constant as a parameter to other routines as opposed to a variable?* The constant parameter may be a flag that is being used to choose among a set of possible computations to perform. A strategy we use for detecting control coupling first computes a set of candidate routines that are invoked with a constant parameter in the library or by code generated from the Amphion domain model. Each member of this set is then analyzed to see if the formal parameter associated with the constant actual parameter is used to conditionally execute disjoint sections of code. We have discovered instances of control coupling in SPICELIB and have computed the set of routines that are invoked with a constant parameter. This set accounts for 19 percent of the total routines in SPICELIB, and we are currently investigating these to see how many of them actually exhibit control coupling.

3.3.3 Reformulation Wrappers

A third heuristic for locating interleaving is to ask: *Which pairs of routines "co-occur"?* Two routines co-occur if they are always called by the same routines, they are executed under the same conditions, and there is a flow of computed data from one to the other. We would like to detect co-occurrence pairs because they are likely to form *reformulation wrappers*. Detecting co-occurrence pairs in the library can heuristically help generate candidates for reformulation wrappers. The pairs need to be checked further to see whether they are inverses of each other. Of course, in general we would like to consider any code fragments as potential pairs, not just library routines.

Through empirical investigation of SPICELIB, we have discovered co-occurrence pairs that form reformulation wrappers and are currently building tools to perform this analysis automatically. The idea is to generate a set of all possible pairs of routines in the system and then eliminate pairs that do not co-occur. Given a pair $\langle S_1, S_2 \rangle$ we say that S_1 and S_2 co-occur if and only if: 1) every subroutine in the library that references S_1 also references S_2, 2) execution of S_1 implies execution of S_2 and vice versa, and 3) there is a flow of computed data from S_1 to S_2. We describe how to implement this test in [31].

4 Open Issues

We are convinced that interleaving seriously complicates understanding computer programs. But recognizing a problem is different from knowing how to fix it. Questions arise as to how powerful tools need to be to detect and extract interleaved components, what form of representation is appropriate to hold the extracted information, how information about the application domain can be used to detect plans, and the extent to which the concept of interleaving and its detection mechanisms scale up to the architectural level.

4.1 Tool Support

We used the Software Refinery from Reasoning Systems in our analyses. This comprehensive toolkit provides a set of language-specific browsers and analyzers, a parser generator, a user interface builder, and an object-oriented repository for holding the results of analyses. We made particular use of two other features of the toolkit. The first is called the Workbench, and it provided pre-existing analyses for traditional graphs and reports such as structure charts, dataflow diagrams, and cross reference lists. The results of the analyses can be accessed from the repository using small, Refine language programs such as those described in [31]. The Refine compiler was the other feature we used, compiling a Refine program into compiled Lisp.

The approach taken by the Refine language and tool suite has many advantages for attacking problems like ours. The language itself combines features of imperative, object-oriented, functional, and rule-based programming, thus providing flexibility and generality. Of particular value to us is its rule-based constructs. Before-and-after condition patterns define the properties of constructs without indicating how to find them. We had merely to add a simple tree walking routine to apply the rules to the abstract syntax tree. In addition to the rule-based features, Refine provides abstract data structures, such as sets, maps, and sequences, which manage their own memory requirements, thereby reducing programmer work. The object-oriented repository further reduces programmer responsibility by providing persistence and memory management.

We also take full advantage of Reasoning Systems' existing Fortran language model and its structure chart analysis. These allowed us a running start on our analysis and provided a robust handling of Fortran constructs that are not typically available from non-commercial research tools.

We can see several ways in which the Refine approach can be extended. In particular, the availability of other analyses, such as control flow graphs for Fortran and general dataflow analysis, would prove useful. Robust dataflow analysis is particularly important to the precision of precondition extraction. The Refine language itself might also be extended further. Currently it does not support passing functions as parameters, and the application of the rule construct is somewhat restricted.

4.2 Representation

The strategy for building program analysis tools is to formulate a *program representation* whose structural properties correspond to interesting program properties. A programming style tool, for example, uses a control flow graph that explicitly represents transfer of execution flow in programs. Irreducible control flow graphs signify the use of unstructured `GO TO` statements. The style tool uses this structural property to report violations of structured programming style. Since we want to build tools for interleaving detection we have to formulate a representation that captures the properties of interleaving. We do this by first listing structural properties that correspond to each of the three characteristics of interleaving and then searching for a representation that has these structural properties.

The key characteristics of interleaving are delocalization, resource sharing, and independence.

In sequential languages like Fortran, delocalization often can not be avoided when two or more plans share data. The components of the plans have to be serialized with respect to the dataflow constraints. This typically means that components of plans cluster around the computation of the data being shared as opposed to clustering around other components of the same plan. This total ordering is necessary due to the lack of support for concurrency in most high level programming languages. It follows then that in order to express a delocalized plan, a representation must impose a partial rather than a total execution ordering on the components of plans.

The partial execution ordering requirement suggests that some form of graphical representation is appropriate. Graph representations naturally express a partial execution ordering via implicit concurrency and explicit transfer of control and data. Since there are a number of such representations to choose from, we narrow the possibilities by noting that:

1. independent plans must be localized as much as possible, with no explicit ordering between them;
2. sharing must be detectable (shared resources should explicitly flow from one plan to another); similarly if two plans p_1, p_2 both share a resource provided by a plan p_3 then p_1 and p_2 should appear in the graph as siblings with a common ancestor p_3);
3. the representation must support multiple views of the program as the interaction of plans at various levels of abstraction, since interleaving may occur at any level of abstraction.

An existing formalism that meets these criteria is Rich's *Plan Calculus* [26, 27, 28]. A plan in the Plan Calculus is encoded as a graphical depiction of the plan's structural parts and the constraints (e.g., data and control flow connections) between them. This diagrammatic notation is complemented with an axiomatized description of the plan that defines its formal semantics. This allows us to develop correctness preserving transformations to extract interleaved plans. The Plan Calculus also provides a mechanism, called *overlays*, for representing correspondences and relationships between pairs of plans (e.g., implementation and optimization relationships). This enables the viewing of plans at multiple levels of abstraction. Overlays also support a very general notion of plan composition which takes into account resource sharing at all levels of abstraction by allowing overlapping points of view.

4.3 Domain Knowledge

Most of the current technology available to help understand programs addresses *how* questions; that is, it is driven by the syntactic structure of programs written in some programming language. But the tasks that require the understanding—perfective, adaptive, and corrective maintenance—are driven by the problem the program is solving; that is, its application domain. These tasks really require answers to *why* questions. For example, if a maintenance task requires extending `NPEDLN` to handle symmetric situations where more than one "nearest point" exist to a line, then the programmer needs to figure out what to do about the distance calculation also computed by `NPEDLN`. Why was `DIST` computed inside of the routine instead of separately? Was it only for efficiency reasons, or might

the nearest point and the distance be considered a *pair* of results by its callers? In the former case, a single DIST return value is still appropriate, in the latter, a pair of identical values is indicated. To answer questions like these, programmers need to know which plans pieces of code are implementing. And plans are direct reflections of the application domain, not the program.

Another example from NPEDLN concerns reformulation wrappers. These plans are inherently delocalized. In fact, they only make sense as plans at all when considered in the context of the application domain: stable computations of solar system geometry. Without this understanding, the best hope is to recognize that the code has uniformly multiplied the elements of a vector in two places, without knowing why this was done and how the multiplications are connected.

The underlying issue is that any scheme for code understanding based solely on a top-down or a bottom-up approach is inherently limited. As illustrated by the examples, a bottom-up approach cannot hope to relate delocalized segments or disentangle interleavings without being able to relate to the plans being implemented. And a top-down approach cannot hope to find where a plan is implemented without being able understand how plan implementations are related syntactically and via data flows. The implication is that a coordinated strategy is indicated, where plans generate expectations that guide program analysis and program analysis generates related segments that need explanation.

4.4 Architectural Interleaving

We can characterize the ways interleaving manifests itself in source code along two orthogonal dimensions. These form a possible design space of solutions to the interleaving problem and can help relate existing techniques that might be applicable. One dimension is the *scope* of the interleaving, which can range from intraprocedural to interprocedural to object to architectural. Another dimension is the *structural mechanism* providing the interleaving, which may be naming, control, data, or protocol. Protocols are global constraints, such as maintaining stack discipline or synchronization mechanisms for cooperating processes. For example, the use of control flags is a control-based mechanism for interleaving with interprocedural scope. The common iteration construct involved in loop fusion is another control-based mechanism, but this interleaving has intraprocedural scope. Reformulation wrappers use a protocol mechanism, usually at the intraprocedural level, but they can have interprocedural scope. Multiple-inheritance is an example of a data-centered interleaving mechanism with object scope.

Interleaving at the scope of objects and architectures or involving global protocol mechanisms is not yet well understood. Consequently, few mechanisms for detection and extraction currently exist in these areas. We believe that more knowledge of the domain will be required to detect interleaving as the scope becomes more global in nature.

5 Related Work

Techniques for detecting interleaving and disentangling interleaved plans are likely to build on existing program comprehension and maintenance techniques.

5.1 The Role of Recognition

When what is interleaved is *familiar* (i.e., stereotypical, frequently used plans), cliché recognition (e.g., [12, 15, 16, 17, 25, 29, 40]) is a useful detection mechanism.[3] In fact, most recognition systems deal explicitly with the recognition of clichés that are interleaved in specific ways with unrecognizable code or other clichés. One of the key features of GRASPR [40], for instance, is its ability to deal with delocalization and redistribution-type function sharing optimizations.

KBEmacs [28, 38] uses a simple, special-purpose recognition strategy to segment loops within programs. This is based on detecting coarse patterns of data and control flow at the procedural level that are indicative of common ways of constructing, augmenting, and interleaving iterative computations. For example, KBEmacs looks for minimal sections of a loop body that have data flow feeding back only to themselves. This decomposition enables a powerful form of abstraction, called *temporal abstraction*, which views iterative computations as compositions of operations on sequences of values. The recognition and temporal abstraction of iteration clichés is similarly used in GRASPR to enable it to deal with generalized loop fusion forms of interleaving. Loop fusion is viewed as redistribution of sequences of values and treated as any other redistribution optimization [40].

Most existing cliché recognition systems tend to deal with interleaving involving *data* and *control* mechanisms. Domain-based clustering, as explored by DM-TAO in the DESIRE system [2], focuses on *naming* mechanisms, by keying in on the patterns of linguistic idioms used in the program, which suggest the manifestations of domain concepts.

Mechanisms for dealing with specific types of interleaving have been explicitly built into existing recognition systems. In the future, we envision recognition architectures that detect not only familiar computational patterns, but also recognize familiar types of transformations or design decisions that went into constructing the program. Many existing cliché recognition systems implicitly detect and undo certain types of interleaving design decisions. However, this process is usually done with special-purpose procedural mechanisms that are difficult to extend and that are viewed as having secondary "supporting roles" to the cliché recognition process, rather than as being an orthogonal form of recognition.

5.2 Disentangling Unfamiliar Plans

When what is interleaved is *unfamiliar* (i.e., novel, idiosyncratic, not repeatedly used plans), other, non-recognition-based methods of delineation are needed. For example, slicing [39, 22] is a widely-

[3] Recognition as a program understanding technique deals with clichés, not plans in general. Only clichéd plans can be recognized, since recognition implies noticing something that is familiar.

used technique for localizing functional components by tracing through data dependencies within the procedural scope. Cluster analysis [2, 13, 32, 33] is used to group related sections of code, based on the detection of shared uses of global data, control paths, and names. However, clustering techniques can only provide limited assistance by roughly delineating possible locations of functionally cohesive components. Another technique, called "potpourri module detection" [6], detects modules that provide more than one independent service by looking for multiple proper subgraphs in an entity-to-entity interconnection graph. These graphs show dependencies among global entities within a single module. Presumably, the independent services reflect separate plans in the code.

Research into automating data encapsulation has recently provided mechanisms for hypothesizing possible locations of data plans at the *object* scope. For example, Bowdidge and Griswold [4] use an extended data flow graph representation, called a star diagram, to help human users see all the uses of a particular data structure and to detect frequently occurring computations that are candidates for abstract functions. Techniques have also been developed within the RE^2 project [7, 8], for identifying candidate abstract data types and their associated modules, based on the call graph and dominance relations. Further research is required to develop techniques for extracting objects from pieces of data that have not already been aggregated in programmer-defined data structures. For example, detecting multiple pieces of data that are always used together might suggest candidates for data aggregation (as for example, in `NPEDLN`, where the input parameters `A`, `B`, and `C` are used as a tuple representing an ellipsoid, and the outputs `PNEAR` and `DIST` represent a pair of results related by interleaved, highly overlapping plans).

6 Conclusion

This paper characterizes interleaving, a particularly troublesome feature of programs which makes them difficult to understand. We offer the following definition:

> *Interleaving* expresses the merging of two or more distinct plans within some contiguous textual area of a program. Interleaving can be characterized by the *delocalization* of the code for the individual plans involved, the *sharing* of some resource, and the implementation of multiple, *independent* plans in the program's overall purpose.

Whether this characterization is robust is an issue for future empirical studies. We plan to study the frequency of occurrence of the various types of interleaving we have identified in our example programs. This may lead to better complexity metrics for determining the maintainability and comprehensibility of programs. Ultimately, designing tools for detection and extraction will be the true test of the usefulness of this characterization.

Acknowledgments Support for this research has been provided by ARPA, (contract number NAG 2-890). We are grateful to JPL's `NAIF` group for enabling our study of their `SPICELIB` software. We also benefited from insightful discussions with Michael Lowry at Nasa Ames Research Center concerning this study and interesting future directions.

7 Appendix: NPELDN with Some of Its Documentation

```
C$ Nearest point on ellipsoid to line.
      SUBROUTINE NPEDLN(A,B,C,LINEPT,LINEDR,PNEAR,
     .                  DIST)
      INTEGER               UBEL
      PARAMETER           ( UBEL = 9 )
      INTEGER               UBPL
      PARAMETER           ( UBPL = 4 )
      DOUBLE PRECISION      A
      DOUBLE PRECISION      B
      DOUBLE PRECISION      C
      DOUBLE PRECISION      LINEPT ( 3 )
      DOUBLE PRECISION      LINEDR ( 3 )
      DOUBLE PRECISION      PNEAR  ( 3 )
      DOUBLE PRECISION      DIST
      LOGICAL               RETURN
      DOUBLE PRECISION      CANDPL ( UBPL )
      DOUBLE PRECISION      CAND   ( UBEL )
      DOUBLE PRECISION      OPPDIR (    3 )
      DOUBLE PRECISION      PRJPL  ( UBPL )
      DOUBLE PRECISION      MAG
      DOUBLE PRECISION      NORMAL (3 )
      DOUBLE PRECISION      PRJEL  ( UBEL )
      DOUBLE PRECISION      PRJPT  (3 )
      DOUBLE PRECISION      PRJNPT (    3 )
      DOUBLE PRECISION      PT     ( 3, 2 )
      DOUBLE PRECISION      SCALE
      DOUBLE PRECISION      SCLA
      DOUBLE PRECISION      SCLB
      DOUBLE PRECISION      SCLC
      DOUBLE PRECISION      SCLPT  (    3 )
      DOUBLE PRECISION      UDIR   (    3 )
      INTEGER               I
      LOGICAL               FOUND  (    2 )
      LOGICAL               IFOUND
      LOGICAL               XFOUND
      IF ( RETURN () ) THEN
         RETURN
      ELSE
         CALL CHKIN ( 'NPEDLN' )
      END IF
      CALL UNORM ( LINEDR, UDIR, MAG )
      IF ( MAG .EQ. 0 ) THEN
         CALL SETMSG('Direction is zero vector.')
         CALL SIGERR('SPICE(ZEROVECTOR)'         )
         CALL CHKOUT('NPEDLN'                    )
         RETURN
       ELSE IF (( A .LE. 0.D0 )
     .          .OR.   ( B .LE. 0.D0 )
     .          .OR.   ( C .LE. 0.D0 )    )   THEN
         CALL SETMSG ('Semi-axes: A=#,B=#,C=#.')
         CALL ERRDP  ('#', A                    )
         CALL ERRDP  ('#', B                    )
         CALL ERRDP  ('#', C                    )
         CALL SIGERR ('SPICE(INVALIDAXISLENGTH)')
         CALL CHKOUT ('NPEDLN'                  )
         RETURN
       END IF
C Scale the semi-axes lengths for better
C numerical behavior.  If squaring any of the
C scaled lengths causes it to underflow to
C zero, signal an error. Otherwise scale the
C point on the input line too.
      SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
      SCLA   =  A / SCALE
      SCLB   =  B / SCALE
      SCLC   =  C / SCALE
      IF (( SCLA**2  .LE.  0.D0 )
     .    .OR.  ( SCLB**2  .LE.  0.D0 )
     .    .OR.  ( SCLC**2  .LE.  0.D0 )    ) THEN
        CALL SETMSG ('Axis too small: A=#,B=#,C=#.')
         CALL ERRDP  ('#', A )
         CALL ERRDP  ('#', B )
         CALL ERRDP  ('#', C )
         CALL SIGERR ('SPICE(DEGENERATECASE)')
         CALL CHKOUT ('NPEDLN' )
         RETURN
      END IF
      SCLPT(1)  =  LINEPT(1) / SCALE
      SCLPT(2)  =  LINEPT(2) / SCALE
      SCLPT(3)  =  LINEPT(3) / SCALE
C Hand off the intersection case to SURFPT.
C SURFPT determines whether rays intersect a body,
C so we treat the line as a pair of rays.
      CALL VMINUS(UDIR, OPPDIR)
      CALL SURFPT(SCLPT, UDIR,    SCLA, SCLB,
     .            SCLC, PT(1,1), FOUND(1))
      CALL SURFPT(SCLPT, OPPDIR, SCLA, SCLB,
     .            SCLC, PT(1,2), FOUND(2))
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
            DIST  =  0.0D0
            CALL VEQU   ( PT(1,I),  PNEAR          )
            CALL VSCL   ( SCALE,    PNEAR,  PNEAR )
            CALL CHKOUT ( 'NPEDLN'                )
            RETURN
         END IF
50001 CONTINUE
C Getting here means the line doesn't intersect
C the ellipsoid. Find the candidate ellipse CAND.
C NORMAL is a normal vector to the plane
C containing the candidate ellipse. Mathematically
C the ellipse must exist; it's the intersection of
C an ellipsoid centered at the origin and a plane
C containing the origin.  Only numerical problems
C can prevent the intersection from being found.
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL (SCLA,SCLB,SCLC,CANDPL,CAND,XFOUND)
      IF ( .NOT. XFOUND ) THEN
         CALL SETMSG ( 'Candidate ellipse not found.')
         CALL SIGERR ( 'SPICE(DEGENERATECASE)'       )
         CALL CHKOUT ( 'NPEDLN'                      )
         RETURN
      END IF
C Project the candidate ellipse onto a plane
C orthogonal to the line.  We'll call the plane
C PRJPL and the projected ellipse PRJEL.
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C Find the point on the line lying in the project-
C ion plane, and then find the near point PRJNPT
C on the projected ellipse.  Here PRJPT is the
C point on the line lying in the projection plane.
C The distance between PRJPT and PRJNPT is DIST.
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT          )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C Find the near point PNEAR on the ellipsoid by
C taking the inverse orthogonal projection of
C PRJNPT; this is the point on the candidate
C ellipse that projects to PRJNPT.  The output
C DIST was computed in step 3 and needs only to be
C re-scaled. The inverse projection of PNEAR ought
C to exist, but may not be calculable due to nu-
C merical problems (this can only happen when the
C ellipsoid is extremely flat or needle-shaped).
      CALL VPRJPI(PRJNPT,PRJPL, CANDPL, PNEAR, IFOUND)
      IF ( .NOT. IFOUND ) THEN
         CALL SETMSG ('Inverse projection not found.')
         CALL SIGERR ('SPICE(DEGENERATECASE)'        )
         CALL CHKOUT ('NPEDLN'                       )
         RETURN
      END IF
C Undo the scaling.
      CALL VSCL ( SCALE,  PNEAR,  PNEAR )
      DIST   =   SCALE * DIST
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
      END
```

```
C Descriptions of subroutines called by NPEDLN:
C
C CHKIN  Module Check In (error handling).
C UNORM  Normalize double precision 3-vector.
C SETMSG Set Long Error Message.
C SIGERR Signal Error Condition.
C CHKOUT Module Check Out (error handling).
C ERRDP  Insert DP Number into Error Message Text.
C VMINUS Negate a double precision 3-D vector.
C SURFPT Find intersection of vector w/ ellipsoid.
C VEQU   Make one DP 3-D vector equal to another.
C VSCL   Vector scaling, 3 dimensions.
C NVC2PL Make plane from normal and constant.
C INEDPL Intersection of ellipsoid and plane.
C PJELPL Project ellipse onto plane, orthogonally.
C VPRJP  Project a vector onto plane orthogonally.
C NPELPT Find nearest point on ellipse to point.
C VPRJPI Vector projection onto plane, inverted.
C -----------------------------------------------
C
C Descriptions of variables used by NPEDLN:
C A        Length of semi-axis in the x direction.
C B        Length of semi-axis in the y direction.
C C        Length of semi-axis in the z direction.
C LINEPT   Point on input line.
C LINEDR   Direction vector of input line.
C PNEAR    Nearest point on ellipsoid to line.
C DIST     Distance of ellipsoid from line.
C UBEL     Upper bound of array containing ellipse.
C UBPL     Upper bound of array containing plane.
C PT       Intersection point of line & ellipsoid.
C CAND     Candidate ellipse.
C CANDPL   Plane containing candidate ellipse.
C NORMAL   Normal to the candidate plane CANDPL.
C UDIR     Unitized line direction vector.
C MAG      Magnitude of line direction vector.
C OPPDIR   Vector in direction opposite to UDIR.
C PRJPL    Projection plane, which the candidate
C          ellipse is projected onto to yield PRJEL.
C PRJEL    Projection of the candidate ellipse
C          CAND onto the projection plane PRJEL.
C PRJPT    Projection of line point.
C PRJNPT   Nearest point on projected ellipse to
C          projection of line point.
C SCALE    Scaling factor.
```

References

[1] V.R. Basili and H.D. Mills. Understanding and documenting programs. *IEEE Trans. on Software Engineering*, 8(3):270–283, May 1982.

[2] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72–83, May 1994.

[3] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[4] R. Bowdidge and W. Griswold. Automated support for encapsulating abstract data types. In *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 97–110, New Orleans, Dec. 1994.

[5] R. Brooks. Towards a theory of the comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18:543–554, 1983.

[6] F. Calliss and B. Cornelius. Potpourri module detection. In *IEEE Conf. on Software Maintenance - 1990*, pages 46–51, San Diego, CA, November 1990. IEEE Computer Society Press.

[7] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data types. In *Proc. of the First Working Conference on Reverse Engineering*, pages 73–82, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[8] A. Cimitile, M. Tortorella, and M. Munro. Program comprehension through the identification of abstract data types. In *Proc. 3rd Workshop on Program Comprehension*, pages 12–19, Washington, D.C., November 1994. IEEE Computer Society Press.

[9] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *GUIDE 48*, 4 1979. Also appears in [24].

[10] R. Hall. Program improvement by automatic redistribution of intermediate results. Technical Report 1251, MIT Artificial Intelligence Lab., February 1990. PhD.

[11] R. Hall. Program improvement by automatic redistribution of intermediate results: An overview. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, Menlo Park, CA, 1991.

[12] J. Hartman. Automatic control understanding for natural programs. Technical Report AI 91-161, University of Texas at Austin, 1991. PhD thesis.

[13] D. Hutchens and V. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. on Software Engineering*, 11(8), August 1985.

[14] Reasoning Systems Incorporated. *Software Refinery Toolkit*. Palo Alto, CA.

[15] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[16] W. Kozaczynski and J.Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1(1):61–78, March 1994.

[17] S. Letovsky. Plan analysis of programs. Research Report 662, Yale University, December 1988. PhD.

[18] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3), 1986.

[19] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: automatic programming for subroutine libraries. In *Proc. 9th Knowledge-Based Software Engineering Conference*, pages 2–11, Monterey, CA, 1994.

[20] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proc. 9th Knowledge-Based Software Engineering Conference*, pages 48–57, Monterey, CA, 1994.

[21] G. Myers. *Reliable Software through Composite Design*. Petrocelli Charter, 1975.

[22] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Comm. of the ACM*, 37(5):50–57, May 1994.

[23] S. Ornburn and S. Rugaber. Reverse engineering: Resolving conflicts between expected and actual software designs. In *IEEE Conf. on Software Maintenance - 1992*, pages 32–40. Orlando, Florida, November 1992.

[24] G. Parikh and N. Zvegintozov, editors. *Tutorial on Software Maintenance*. IEEE Computer Society, 1983. Order No. EM453.

[25] A. Quilici. A memory-based approach to recognizing programming plans. *Comm. of the ACM*, 37(5):84–93, May 1994.

[26] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981.

[27] C. Rich. Inspection methods in programming. Technical Report 604, MIT Artificial Intelligence Lab., June 1981. PhD thesis.

[28] C. Rich and R. C. Waters. *The Programmer's Apprentice*. Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.

[29] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990. Reprinted in P. H. Winston, editor, *Artificial Intelligence at MIT: Expanding Frontiers*, MIT Press, Cambridge, MA, In press.

[30] S. Rugaber, S. Ornburn, and R. LeBlanc. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[31] S. Rugaber, K. Stirewalt, and L. Wills. Detecting interleaving. In *IEEE Conf. on Software Maintenance - 1995*, Nice, France, September 1995. IEEE Computer Society Press. To appear.

[32] R. Schwanke. An intelligent tool for re-engineering software modularity. In *IEEE Conf. on Software Maintenance - 1991*, pages 83–92, 1991.

[33] R. Schwanke, R. Altucher, and M. Platoff. Discovering, visualizing, and controlling software structure. In *Proc. 5th Int. Workshop on Software Specs. and Design*, pages 147–150, Pittsburgh, PA, 1989.

[34] P. Selfridge, R. Waters, and E. Chikofsky. Challenges to the field of reverse engineering - A position paper. In *Proc. of the First Working Conference on Reverse Engineering*, pages 144–150, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[35] D. Smith, G. Kotik, and S. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Trans. on Software Engineering*, November 1985.

[36] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10(5):595–609, September 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[37] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood, and A. Bundy. Deductive composition of astronomical software from subroutine libraries. In *Proc. 12th International Conference on Automated Deduction*, pages 341–55, Nancy, France, 1994.

[38] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[39] Mark Weiser. Program slicing. In *5th Int. Conf. on Software Engineering*, pages 439–449, San Diego, CA, 3 1981.

[40] L. Wills. Automated program recognition by graph parsing. Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD Thesis.

[41] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.

Detecting Interleaving

Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{spencer, kurt, linda}@cc.gatech.edu

Abstract

The various goals and requirements of a system are realized in software as fragments of code that are typically "interleaved" in that they may be woven together in the same contiguous textual area of code. The fragments of code are often delocalized and overlap rather than being composed in a simple linear sequence. Interleaving severely complicates software comprehension and maintenance. To address this problem, we are developing analysis tools, based on the Software Refinery. This paper describes our experiences in detecting interleaving in a corpus of mathematical software written in Fortran from the Jet Propulsion Laboratory. In particular, it discusses how feasible it is to detect interleaving of various types and the ability of existing tools to assist these types of detection.

1 Introduction and Motivation

To understand a program, one often has to unravel multiple, interwoven strands of computation, each responsible for accomplishing a distinct purpose. We use the term *plan* to denote a description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program [13, 19]. Plans can occur at any level of abstraction from architectural overviews to code. Interleaving expresses the merging of two or more distinct plans within some contiguous textual area of a program.

A trivial example is a single loop responsible for computing both the maximum element of a vector and its position. A less trivial example is a program intended to write a report that summarizes data extracted from sorted input records. The program has two purposes: computing the summary data and managing the construction of the report (headers, page breaks, page counts, etc.) In an object-oriented program, these two purposes might well be realized by two separate objects. In traditional code, however, the implementations of these functions are often interleaved, and understanding the code is significantly complicated.

Interleaving may arise for efficiency reasons. For example, it may be more efficient to compute two related values in one place than to do so separately. Or interleaving may be the result of inadequate software maintenance, such as adding a feature locally to an existing routine rather than undertaking a thorough redesign. Or interleaving may arise as a natural by-product of expressing separate but related plans in a linear, textual medium. For example, accessors and constructors for manipulating data structures are typically interleaved throughout programs written in traditional programming languages due to their procedural, rather than object-oriented structure. Regardless of why interleaving is introduced, it severely complicates understanding a program. This makes it difficult to perform a variety of tasks, including extracting reusable components, localizing the effects of maintenance changes, and migrating to object-oriented languages.

What is needed is to isolate the separate strands of computation, understand them individually and then see how they interrelate. We are building analysis tools to do this. Traditional slicing techniques [25] are applicable to this problem but are not powerful enough to disentangle all forms of interleaving. They rely only on data and control flow; whereas our tools also use deeper knowledge of what plans are interleaved in the code.

1.1 Case Study Context

Software must be understood at multiple levels of abstraction simultaneously. Plans can occur and be interleaved at any abstraction level from source code text to application domain structures. The process of

understanding a piece of software involves two parallel knowledge acquisition activities [4, 16, 23]:

1. using domain knowledge to understand the code - knowledge about the application sets up expectations about *how* abstract concepts are typically manifested in concrete code implementations;

2. using knowledge of the code to understand the domain - what is discovered in the code is used to build up a description of various aspects of the application and help to answer questions about *why* certain code structures exist and what is their purpose with respect to the application.

We are studying interleaving in the context of performing these activities. In particular, we have a corpus of software and an incomplete model of the software's application domain. We are targeting our detection of interleaving toward elaborating the existing domain model for this software. We are also looking for ways in which the current knowledge in the domain model can guide detection and ultimately comprehension.

SPICELIB

Our corpus of software is a library, called SPICELIB, of approximately 600 mathematical programs, written in Fortran at the Jet Propulsion Laboratory (JPL) for analyzing data sent back from space missions. The software performs calculations in the domain of solar system geometry, such as coordinate frame conversions, intersections of rays, ellipses, planes, and ellipsoids, and light-time calculations.

Amphion Domain Model

We have obtained a partial model of the application domain of this software from NASA Ames researchers, who have developed a component-based software synthesis system, called Amphion [14]. Amphion composes routines from SPICELIB by making use of a domain theory that includes formal specifications of the library routines, connecting them to abstract concepts in the solar system geometry domain. The knowledge of the domain is encoded in a structured representation, expressed as axioms in first-order logic with equality. A space scientist using Amphion can schematically specify the geometry of a problem through a graphical user interface, and Amphion automatically generates Fortran programs to call library routines to solve the described problem. Amphion is able to do this by proving a theorem about the solvability of the problem in the domain and, as a side effect, generating the appropriate code.

In collaboration with NASA Ames researchers, we are detecting ways in which the domain model Amphion uses is incomplete and developing program comprehension techniques to extend it.

Some of the primary motivations for doing this from the synthesis perspective are to make component retrieval more accurate, to assist in updating and growing the domain model as new software components are added (perhaps extracted from legacy software), and to improve the software synthesized. From the program comprehension perspective, the refinement and elaboration of domain knowledge, based on what is discovered in the code, is a primary activity, driving the generation of hypotheses and informing future analyses.

There are at least two ways in which the existing domain model is incomplete that are particularly interesting from the point of view of interleaving. One incompleteness is that the model does not fully cover the functionality of the routines. Some routines compute more than one result (e.g., the nearest point on an ellipsoid to a line and the shortest distance between that point and the ellipsoid). However, the domain model does not always capture all the values that are computed. In these routines, it is often the case that the code responsible for the secondary functionalities is interleaved with the code for the primary function covered by Amphion's domain model. A second example of incompleteness is that the current domain model does not capture preconditions on the use of the library routines (e.g., that a line given as input to a routine is not the zero vector or that an ellipsoid's semi-axes must be large enough to be scalable). The code responsible for checking these preconditions is usually tightly interleaved with the code for the primary computation as it is sprinkled throughout the routine and often uses intermediate results computed for the primary computation.

Software Refinery

We are developing mechanisms for detecting various classes of interleaving with the aim of growing a partial model of the application domain. We are building these mechanisms on a collection of commercial tools, called the Software Refinery [11]. This is a comprehensive tool suite including language-specific analyzers and browsers for Fortran, C, Ada, and Cobol, language extension mechanisms for building new analyzers, and a user interface construction tool for displaying the results of analysis. It maintains an object-

oriented repository for holding the results of analyses, such as abstract syntax trees and symbol tables. It provides a powerful wide-spectrum language, called Refine [22], which supports pattern matching and querying the repository. Using the Software Refinery allows us to leverage commercially available tools as well as evaluate the strengths and limitations of its approach to program analysis.

1.2 Contributions and Outline of Paper

We have developed an initial characterization of interleaving [20], based primarily on an empirical study of SPICELIB. We briefly summarize our characterization in Section 2. We then present a detailed example of one of the mechanisms we have built to detect a particular type of interleaving – the interleaving of exception-handling code with the program's primary computation. This simple but pervasive type of interleaving is particularly useful to detect and extract for the purposes of elaborating specifications with preconditions. In Section 4, we describe a set of analyses that we have formulated to detect various types of interleaving, in addition to exception handling. We discuss the results we have obtained so far in implementing these analyses using the Software Refinery and predict what is needed to carry out the rest. In Section 5, we reflect on our experiences in using the Software Refinery to build interleaving detection mechanisms and discuss future directions we would like to explore in detecting and extracting interleaving.

2 Characterizing Interleaving

Interleaving expresses the merging of two or more distinct plans within some contiguous textual area of a program. Interleaving can be characterized by the *delocalization* of the code for the individual plans involved, the *sharing* of some resource, and the performance of multiple, *independent* roles in the program's overall purpose. (Our characterization of interleaving and several detailed examples are presented in more depth in [20].)

There are several reasons why interleaving is a source of difficulties. The first has to do with delocalization. Because two or more design purposes are implemented in a single segment of code, each individual code fragment responsible for a separate purpose is more spread out than it would be if it were encapsulated. This makes it difficult to gather together all the pieces to ensure consistent maintenance [13]. Distracting details also get in the way and must be filtered out from the midst of the delocalized plan.

Another reason why interleaving presents a problem is that it may be the result of poorly thought out maintenance activities, where the original, highly coherent structure of the system has degraded as "patches" and "quick fixes" are introduced.

There may also be occasions where interleaving is intentionally introduced, such as for purposes of optimization. But expressing intricate optimizations in a clean and well-documented fashion is not typically done. The sharing of some resource is characteristic of intentional interleaving. When interleaving is introduced into a program, there is normally some implicit relationship between the interleaved plans, motivating the designer to choose to interleave them. Often this relationship centers on some common resource, such as intermediate data results, control flags, and lexical module structures. This causes the implementations of the interleaved plans to overlap in that a single structural element contributes to multiple goals. In general, the difficulty that resource sharing introduces is that it causes ambiguity in interpreting the purpose of program pieces. This can lead to incorrect assumptions about what effect changes will have, since the maintainer might be focusing on one of the actual uses of the resource (variable, value, control flag, data structure slot, etc.).

While interleaving is introduced to take advantage of commonalities, the flip side of the coin is that the interleaved plans each have a distinct purpose. Although interleaving is necessary for efficiency, it obscures the independence of the components involved. Ironically, this hinders activities like parallelization and objectivization that improve the efficiency and reusability of the code.

For all of these reasons, our ability to comprehend code containing interleaved fragments is compromised. Hopefully, we can have more success by isolating the separate concerns, understanding them individually, and only then seeing how they relate.

3 Extraction of Preconditions

Using the Software Refinery, we have been able to automate a number of program analyses, one of which is the detection of subroutine parameter precondition checks. Because precondition checks are often interspersed throughout a subprogram, they tend to delocalize the plans that perform the primary computational work. In particular, they are usually part of a larger plan that detects exceptional (usually erroneous) conditions in the state of a running program, and takes alternative action when these conditions arise (such as returning with an error code, signalling, or invoking error handlers).

Precondition checks make explicit the assumptions a subprogram places on its inputs. Ideally a tool to

```
C$Procedure SURFPT ( Surface point on an ellipsoid )
      SUBROUTINE SURFPT ( POSITN, U, A, B, C,
     .                    POINT, FOUND )
      DOUBLE PRECISION U      ( 3 )
      ...declarations...
C     Check the input vector to see if its the zero
C     vector. If it is signal an error and return.
C
      IF  ( ( U(1) .EQ. 0.0D0 ) .AND.
     .      ( U(2) .EQ. 0.0D0 ) .AND.
     .      ( U(3) .EQ. 0.0D0 )        ) THEN
         CALL SETMSG (
     .     'SURFPT: Input vector is zero vector.' )
         CALL SIGERR ( 'SPICE(ZEROVECTOR)' )
         CALL CHKOUT ( 'SURFPT' )
         RETURN
      END IF
      ...
```

Figure 1: A fragment of the subroutine SURFPT in SPICELIB. This fragment shows a precondition check which invokes an exception if all of the elements of the U array are 0.

aid in program understanding removes the interleaved precondition checks from the code and uses the information they represent to help the user *elaborate* a high-level specification of the subprogram. Such a tool addresses program understanding by extracting an interleaved plan, and by assisting in the composition of a specification by suggesting subprogram preconditions. We have created a tool that detects these checks and extracts the preconditions into a documentation form suitable for expression as a partial specification.

We found numerous examples of precondition checks in our empirical analysis of the SPICELIB. One such check occurs in the subprogram SURFPT shown in Figure 1. SURFPT finds the intersection (POINT) of a ray (represented by a point POSITN and a direction vector U) with an ellipsoid (represented as three semi-axes lengths A, B, and C), if such an intersection exists (indicated by FOUND).

Precondition checks are particularly difficult to understand when they are sprinkled throughout the code of a subroutine as opposed to being localized at the beginning. We discovered that, though interleaved, these checks could be reliably identified by searching for IF statements whose conditions are a function of the input parameters and whose bodies handle exceptions. The analysis that decides whether or not IF statements test only input parameters is specific to the Fortran language; whereas the analysis that decides if a code fragment is an exception plan is application domain specific. The implication of this is that the Fortran specific portion is not likely to need changing when we apply the tool to a new application; whereas the application specific portion will certainly need to change. With this in mind, we chose a tool architecture that gives a great deal of freedom in the expression of the procedure for detecting exception plans.

Detecting Exception Handlers In general, we need application specific knowledge about usage patterns in order to discover exception handlers. SPICELIB, for example, provides a routine SIGERR that sets an error condition. Typically, SIGERR is followed almost immediately by a RETURN statement. Hence, a call to SIGERR followed closely by a RETURN indicates a plan for handling an exception. In some other application, the form of this plan will be much different. It is, therefore, necessary to design the plan detection component of our architecture around this need to specialize the tool with knowledge about the application of a system being analyzed.

The Software Refinery provides excellent support for this design principle through the use of the **rule** construct and a tree-walker that applies these rules to an abstract syntax tree (AST). Briefly, a rule is a construct in the Refine language that encapsulates a transformation of the state of the system (that is, a side effect). Rules are specifically designed to be applied to the nodes of an AST during tree walks, so they always have a single input parameter that is some subclass of AST nodes. Rules are more declarative than functions in that they specify state changes by listing the *conditions* before and after the change without specifying exactly how the change must occur. Syntactically, this is expressed through the form: *preconditions* $\rightarrow$ *postconditions*. This is useful for linking application specific pattern knowledge into a system because it allows the independent, declarative expression of the different facets of the pattern.

We represent application specific exception handler plan clues using two rules. The first searches for a RETURN statement in an AST:

```
rule IS-RETURN ( s : rf::program-unit-statement )
   ~*FOUND-RETURN* &
   rf::return-statement(s)
   -->
   *FOUND-RETURN*
```

This rule states that if, during an AST walk, a RETURN statement has not yet been discovered (represented by the negating the value of the boolean global variable

FOUND-RETURN) and the current AST node (bound by the parameter s) in the tree walk is of class rf::return-statement, then establish that a RETURN has been found (similarly represented by the global variable *FOUND-RETURN*). The careful reader may note that the first conjunct in the precondition of this rule is not logically necessary (~*FOUND-RETURN*). It is included here because rule application in Refine continues until no more rules can be matched on any AST node. So rules must somehow invalidate their precondition when they are applied or else the rule application will go into an infinite loop.

The other rule we use to analyze SPICELIB is:

```
rule IS-SIGERR ( s : rf::program-unit-statement )
   ~*FOUND-SIGERR* &
   rf::call-statement(s) &
   rf::identifier-name(rf::called-object(s)) =
                                   'RFU::SIGERR
   -->
   *FOUND-SIGERR*
```

This rule is similar to the rule to detect RETURN, but in this case we are checking not only that the current AST node (s) belongs to a certain class (rf::call-statement), but also that the name of the called object is SIGERR. The code that applies these rules to sequences of statements is shown in Figure 2. Note here the application of the function preorder-transform which applies a sequence of rules to an AST until no more rules apply. We specify the actual sequence of rules to apply by defining the variable *CHECK-EXCEPTION-RULES* as follows:

```
var *CHECK-EXCEPTION-RULES* : seq(symbol) =
  [ 'IS-RETURN,
    'IS-SIGERR ]
```

That is, *CHECK-EXCEPTION-RULES* is a variable whose type is a sequence of symbols, assigned to the literal sequence of symbols denoting the rules we have defined.

Detecting Guards Discovering IF statements that depend only upon input parameters (guards) involves keeping track of whether or not these parameters have been modified before the check. If they have been modified before the check, then the check probably is not a precondition check on inputs. We address this problem in our analysis by using an approximate dataflow algorithm which propagates a set of immutable variables through the sequence of statements in the subroutine. (We do this in the absence of a robust and complete dataflow analysis tool.) At each

```
"Given a sequence of Fortran statements
 (assumed to be the body of an IF-THEN-ELSE
 block), report whether or not the statements
 appear to be raising an exception."
function Looks-Like-Exception(
        stmt-seq : seq(program-unit-statement) ) :
boolean =
   let (*FOUND-RETURN* : boolean = false,
        *FOUND-SIGERR* : boolean = false)
      (enumerate a-stmt over stmt-seq do
         preorder-transform(
               a-stmt,*CHECK-EXCEPTION-RULES*));
      *FOUND-RETURN* & *FOUND-SIGERR*
```

Figure 2: Refine code that checks for exception plans by applying the pattern our initial analysis allowed us to deduce.

statement, if a variable X in the set might be modified by the execution of the statement, then X is removed from the propagating set. We define a Refine function Propagate-Through-Statement that does this propagation on a per statement basis. The function is used in our function Get-Preconditions shown in Figure 3.

Intuitively, Get-Preconditions computes the parameters of its input subr using Refine/Fortran primitives and then puts these parameters into a working set worklist that is propagated through statements in the subprogram. The local variable candidates is used to store the candidate preconditions found during a propagation. When an IF-THEN-ELSE statement is found, candidates is updated to be the concatenation of its previous value and a new sequence of preconditions. The function Match returns these sequences of preconditions, returning the empty sequence if the given IF statement did not match our pattern.

Results The result of this analysis is a table of preconditions associated with each subroutine. Since we are targetting partial specification elaboration, we chose to make the tool output the preconditions in LaTeX form so as to generate nicely formatted reports. We include an example here of the preconditions for the subroutine SURFPT. When applied to SURFPT our tool generated the LaTeX source which when included without change into this document looks like:

$$\neg((U(1) = 0.0D0) \wedge (U(2) = 0.0D0) \wedge (U(3) = 0.0D0))$$

Taken literally, this states that one of the first three elements of the U array parameter must be non-zero. In the domain model, U is seen as a vector, so the

```
function Get-Preconditions( subr : program-unit ) :
condition-tuple-type =
  let ( parameters : set(symbol) =
          { Get-Parameter-Name(p) |
            (p) p in formals(unit-heading(subr)) })
    let (worklist   : set(symbol) = params,
         candidates : condition-tuple-seq = [])
      (enumerate s over unit-body(subr) do
         (if block-if-statement(s)
          then candidates <-
                  concat(candidates,
                         Matches(s, worklist)));
        worklist <- Propagate-Through-Statement(
                              s,worklist));
       candidates
```

Figure 3: Given the AST representation of a Fortran subprogram, compute a candidate set of preconditions for the subprogram.

more abstract precondition can be stated as "U is not the zero vector." Extracting the precondition into the literal representation is the first step to being able to express the more abstract precondition.

4 Experiences

We formulated a number of other interleaving analyses in addition to precondition detection. Each answers some empirical question related to interleaving in code libraries. Since some depend upon application domain information, we took advantage of the information encoded in the Amphion domain axioms. We were, however, careful to design the tools so as not to restrict their applicability to this particular domain. We have used our experience with the Software Refinery to implement some of these analyses and to estimate the difficulty of implementing the others.

Indefinite Loops and Dataflow Analysis Precision. Most of the analyses for detecting interleaving rely on some form of dataflow analysis. Dataflow analysis propagates data usage information along all possible *sequences* of statements in a subprogram. The accuracy or precision of a dataflow analysis depends upon the ability to cover all such sequences. Unfortunately, programs with loops can sometimes represent *indefinite* sequences of statements. When posed with such programs we must settle for only approximate dataflow information. On the other hand, subprograms whose loops are bounded by constants may be *unrolled* into code each of whose statement sequences may be determined statically. Such subprograms can be analyzed exactly with dataflow techniques. Loops of this nature occur frequently in SPICELIB. For example, because three-dimensional vectors are stored as arrays of length three, it is typical to see many DO loops with only three iterations.

Since so many of our interleaving analyses depend upon the precision of dataflow analyses, we would like some measure of a tool's maximum analysis potential over the library. We chose to measure this as the percentage of routines with only definite (statically determinable bounds) loops and developed a statistics gathering tool in Refine. The tool works by defining rules to detect indefinite loops and then applying these rules through an AST walk similar to that of the precondition detection tool.

The analysis showed that roughly 68 percent of the subprograms had no indefinite loops. This number indicates that we will be able to completely analyze two thirds of the routines. Moreover, in the other cases, approximate dataflow information may be good enough to capture the spirit of the analysis.

Routines with Multiple Outputs. Some subroutines in SPICELIB compute more than one output. When this occurs, the subroutine is returning either the results of multiple distinct computations or a result whose type can not be directly expressed in the Fortran type system (e.g., as a data aggregate). In the former case, the subroutine is realized as the interleaving of multiple distinct plans. This interleaving not only complicates the task of understanding the code, but also clouds a maintainer's conceptual categorization of the subroutine.

In the latter case, the subroutine may be implementing only a single plan, but a maintainer's conceptual categorization of the subroutine is still obscured by the appearance of some number of seemingly *distinct* outputs. A good example of this case occurs in the SPICELIB subroutine SURFPT:

```
SUBROUTINE SURFPT (POSITN,U,A,B,C,POINT,FOUND)
```

which conceptually returns the intersection of a vector with the surface of an ellipsoid. However, it is possible to give a vector and an ellipsoid that do not intersect. In such a situation the output parameter POINT will be undefined, but the Fortran type system cannot express the type: DOUBLE PRECISION $\vee$ *Undefined*. The programmer was forced to simulate a variable of this type using two variables, POINT and FOUND, adopting the convention that when FOUND is **false**, the return value is *Undefined*, and when FOUND is **true**, the return value is POINT.

Clearly subprograms with multiple outputs complicate program understanding. We built a tool that determines the multiple output subprograms in a library by analyzing the *direction* of dataflow in parameters of functions and subroutines. A parameter's direction is either: **in** if the parameter is only read in the subprogram, **out** if the parameter is only written in the subprogram, or **in-out** if the parameter is both read and written in the subprogram. Multiple output subprograms will have more than one parameter with direction out or in-out.

Fortunately, the Software Refinery provides a package to create structure chart (call graph) objects. The nodes of these structure charts are annotated with direction information about parameters. We were thus able to build our entire analysis using only the structure chart object without having to do *any* Fortran AST analysis. This greatly simplified the implementation.

The resulting analysis showed that 25 percent of the subprograms had multiple output parameters. We were thus able to focus our work on these routines first, as they are likely to involve interleaving.

Domain Model Coverage. NASA has developed a formal model of SPICELIB's application domain. This model is incomplete with respect to the library. One analysis that would focus efforts to complete the domain model stems from a notion of *coverage*. We say a subprogram is an element of the cover induced by a domain model if it is either directly linked to some entity in the domain model or is invoked by another subprogram that is in the cover. A domain model covers a library if every subprogram in the library is in the domain model's cover.

We constructed a tool that determines the set of subroutines in the cover of a domain model. The rich choice of data structures in Refine significantly eases this task. The major constituent of the tool converts a structure chart into a relation that maps subprograms to the subprograms that they call and then computing the transitive closure of this relation. Since Refine provides a transitive closure operation (`tclosure`) in the language, this was a simple task. The result of this analysis was that only 35 percent of the library was covered by the domain model.

Dead End Dataflows. Many of the SPICELIB subprograms that exhibit interleaving also exhibit a behavior detectable in the domain model, called a "dead end dataflow." It occurs when the domain model references a subprogram in the library, and this subprogram returns a value that is not mapped to anything in the domain model. For example, a SPICELIB subprogram that computes the nearest point on an ellipsoid to a line also computes the shortest distance between the line and the ellipsoid, but only the near point output is mapped to a domain entity (the nearest point); the distance output is a dead end dataflow. Dead end dataflows imply interleaving in the subprogram and/or an incompleteness in the domain model. Our analysis revealed that of the subroutines covered by the domain model, 30% have some output parameters that are dead end dataflows.

Control Coupling. Sometimes a programmer will implement a subprogram that uses one of its input parameters as a flag to choose among a set of possible computations to perform. This is a form of control coupling [27]: "any connection between two modules that communicates elements of control," and it is a class of interleaving involving delocalized control. Subprogram calls with constant parameter values are potentially involved in control coupling. Our strategy for detecting control coupling instances first computes a set of candidate subprograms that are invoked with a constant parameter in the library or the domain model. Each member of this set is then analyzed to see if the formal parameter associated with the constant actual parameter is used to conditionally execute disjoint sections of code.

We have not yet implemented this analysis, but the two components of our strategy should be easy to implement in Refine. In addition to applying this information to the elaboration of specifications, we are also interested in seeing if the strategy is significantly more precise than the following variant. It seems likely that if every call of a subprogram passes a constant parameter, then that subprogram is involved in control coupling interleaving. We expect to test this variant along with the original analysis to see if there is a correspondence of results.

Routine Co-occurrence Properties. In [20] we described a general form of interleaving called *reformulation wrappers*. A reformulation wrapper is used to transform one problem into another that is simpler to solve and then to transfer the solution back to the original situation. Some examples of reformulation wrappers in SPICELIB are: reducing a three-dimensional geometry problem to a two-dimensional one and mapping an ellipsoid to the unit sphere to make it easier to solve three-dimensional intersection problems.

Often reformulation wrappers are realized as two functions: one that transforms a problem into a different (usually easier or more stable) problem, and one that transforms the result back. We would like to detect instances of this in arbitrary codes, and we believe that the key to this detection is embodied in a notion of subprogram *co-occurrence*. We say that two subprograms S_1 and S_2 in a library co-occur if: (1) every subroutine in the library that references S_1 also references S_2, (2) execution of S_1 implies execution of S_2 and vice versa, and (3) there is flow of computed data from S_1 to S_2.

This is by far the most ambitious interleaving analysis we have mentioned here. Its solution requires substantial dataflow infrastructure. To get a feel for this requirement, we examine each point in the definition.

Given a pair (S_1, S_2), it is relatively simple to determine if each subprogram in the library references both of them. Refine provides language constructs for specifying first order logic predicates. Using this feature of the language, we could implement this test as:

```
fa(x)(x in *SPICELIB* =>
        (contains(x,s1) => contains(x,s2)) &
        (contains(x,s2) => contains(x,s1)))
```

and define the function `contains` to return whether or not the subprogram in the first parameter contains a call to the subprogram in the second parameter. It should be obvious from our other examples that the `contains` function is simple to construct in Refine.

We can express the execution conditions regarding S_1 and S_2 using a construct from flow analysis called a dominator [1]. We say that the call of S_1 *dominates* the call of S_2 if the call of S_2 implies that S_1 has already been called in that routine. Similarly, the call of S_2 *post-dominates* the call of S_1 if calling S_1 implies there will be a call to S_2 in that routine. Refine does not provide an analysis for dominators, but we have built our own from other Refine packages.

To complete the analysis of reformulation wrappers, we still need to check that data computed in S_1 somehow flows into the inputs of S_2. To compute this, we need a full dataflow analysis component. Refine does not provide such a component, but we are working on one of our own.

5 Conclusions

Interleaving is a pervasive and troubling problem. Disentangling interleaved program strands can improve understandability and provide opportunities for reuse, thus extending the value and lifetime of software assets. Detecting and extracting interleaved strands is a complex problem that we are just beginning to understand. This section describes the strengths and weaknesses of available tools and the successes and failures we have had in understanding instances of interleaving. We also comment on the role of domain information in understanding interleaved code and our plans for continuing to explore this area.

5.1 Tools

We used the Refine tools from Reasoning Systems in our analysis. This comprehensive toolkit provides a set of language-specific browsers and analyzers, a parser generator, a user interface builder, and an object-oriented repository for holding the results of analysis. We made particular use of two other features of the toolkit. The first was called the Workbench, and it provided pre-existing analyses for traditional graphs and reports such as structure charts, dataflow diagrams, and cross reference lists. The results of the pre-existing analyses can be accessed from the repository using small, Refine language programs such as those described in this paper. The Refine compiler was the other feature we used, compiling a Refine program into compiled Lisp.

The approach taken by the Refine language and tool suite has many advantages for attacking problems like ours. The language itself combines features of imperative, object-oriented, functional, and rule-based programming, thus providing flexibility and generality. Of particular value to us is its rule-based constructs. By merely defining pre- and post-conditions we are easily able to define the properties of constructs without worrying about how to find them. We had merely to add a simple tree walking routine to apply the rules to the AST. Furthermore, the language provides abstract data structures, such as sets, maps, and sequences, which manage their own memory requirements, thereby reducing programmer work. The object-oriented repository further reduced programmer responsibility by providing persistence and memory management.

We also take full advantage of Reasoning Systems' existing Fortran language model and its structure chart analysis. These allowed us a running start on our analysis and also provided a robust handling of Fortran constructs that are not typically available from non-commercial research tools. Finally, we had occasion to take advantage of the existing Refine user community through its mailing list (`refine-users@grace.rt.cs.boeing.com`). Our technical questions were answered nearly instantaneously and always accurately.

We can see several ways in which the Refine approach can be further extended. In particular, the

availability of other analyses, such as control flow graphs for Fortran and general dataflow analysis, would have given us more leverage. The Refine language itself might be extended further, such as by supporting function parameters and more transparent rule application.

In summary, the Refine approach and tool suite provided extensive leverage without which we would not have been able to adequately explore the interleaving question.

5.2 Interleaving

In Section 2, we characterized interleaving in terms of delocalization, resource sharing, and independence. The particular analyses we performed can be viewed in terms of these dimensions. For example, precondition checks are often spread throughout (delocalized in) a routine. The same holds true for reformulation wrappers. In the absence of full dataflow analysis, we approximated the actual analysis we would like to have made, albeit in a safe way. That is, all of the results produced were accurate, but some additional ones may have been missed due to missing information about dataflow relationships. However, we expect these to be few in number.

Resource sharing and independence are related concepts. There must be some reason, usually expressible as a shared resource, for two independent fragments to be interleaved. For example, routines with multiple outputs share intermediate computations as a way of improving run-time efficiency. This particular kind of interleaving was also easily detected.

In general, the detection and extraction of interleaved fragments is a hard problem. It is related to *overlapping implementations* [18], *potpourri module detection* [5], *slicing* [25, 15], *cluster analysis* [2, 10, 21], and *objectivization* (the extraction of candidate objects from non-object oriented programs) [3, 6, 7]. To the extent that the analysis can be based on program information only, the approach we have undertaken appears adequate. However, interleaved fragments are instantiations of plans, and many plans are direct expressions of application domain requirements. Hence, extensive use of domain knowledge may be a prerequisite to complete analysis.

5.3 Domains

In our analysis, we had the benefit of an existing partial domain model. The domain model provides expectations and data to be used in analyzing a program. For example, there were several readily distinguishable instances in the domain model of pairs of routines that we used as candidates for detection of reformulation wrappers. On the other hand, we have been able to use the results of program analysis to validate and extend the domain model. For example, there were situations where a routine computed several results, but the domain model only described one of them.

The simultaneous elaboration of application and program understanding is characteristic of Synchronized Refinement, the overarching approach we take to reverse engineering [16]. How best to express and use the domain information is still an open question, however, but some discussion of the issues is provided in [8]. We expect the use of domain information to be a prerequisite for further significant increases in the power of program understanding technology.

5.4 Future Directions

The work described here raises several further questions related to interleaving. One of these concerns the appropriate level of abstraction in the domain model. In our case, the domain model that we used was fairly low level. If the ultimate goal is to map a program back to its requirements, then, program constructs must be described in terms of domain vocabulary. For example, if we can detect certain stereotypical constructs (called *clichés* [19]) and map them back to a domain concept (such as an ellipsoid or an orthogonal projection), then we can use our knowledge of geometry to further understand and describe a program. This requires the use of cliché recognition techniques (e.g., [12, 17, 26]). It also raises the question of describing the domain itself in Refine, and consequently being able to reason about it.

We would also like to look at architectural issues. In particular, the analyses we performed were fairly low level, and, in fact, the SPICELIB software itself has a fairly straightforward architecture. But this will not always be true. In fact, architectures can be interleaved just like programs. This problem appears to be quite hard, but we can see attacking it both from the top down, by trying to detect instances of specific architectural styles [9], and from the bottom up, by trying to detect specific kinds of module groupings, such as the afferent, efferent, and transformer modules that are a part of Structured Design [24].

Acknowledgments

Support for this research has been provided by ARPA under order number A870, contract number NAG 2-890. We would like to thank Larry Markosian for helping us to obtain the Software Refinery and the NAIF group at JPL for enabling our study of SPICELIB. We also benefited from insightful discussions with Michael Lowry at Nasa Ames Research Center concerning this research.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72-83, May 1994.

[3] R. Bowdidge and W. Griswold. Automated support for encapsulating abstract data types. In *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 97-110, New Orleans, Dec. 1994.

[4] R. Brooks. Towards a theory of the comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18:543-554, 1983.

[5] F. Calliss and B. Cornelius. Potpourri module detection. In *IEEE Conf. on Software Maintenance - 1990*, pages 46-51, San Diego, CA, November 1990. IEEE Computer Society Press.

[6] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data types. In *Proc. of the First Working Conference on Reverse Engineering*, pages 73-82, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[7] A. Cimitile, M. Tortorella, and M. Munro. Program comprehension through the identification of abstract data types. In *Proc. 3rd Workshop on Program Comprehension*, pages 12-19, Washington, D.C., November 1994. IEEE Computer Society Press.

[8] J-M. DeBaud, B. Moopen, and S. Rugaber. Domain analysis and reverse engineering. In *IEEE Conf. on Software Maintenance - 1994*, pages 326-335.

[9] A. Garlan and M. Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[10] D. Hutchens and V. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. on Software Engineering*, 11(8), August 1985.

[11] Reasoning Systems Incorporated. *Software Refinery Toolkit*. Palo Alto, CA.

[12] W. Kozaczynski and J.Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1(1):61-78, March 1994.

[13] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3), 1986.

[14] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Knowledge-based Software Engineering Conference*, 1994.

[15] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Comm. of the ACM*, 37(5):50-57, May 1994.

[16] S. Ornburn and S. Rugaber. Reverse engineering: Resolving conflicts between expected and actual software designs. In *IEEE Conf. on Software Maintenance - 1992*, pages 32-40, Orlando, Florida, November 1992.

[17] A. Quilici. A memory-based approach to recognizing programming plans. *Comm. of the ACM*, 37(5):84-93, May 1994.

[18] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044-1052, Vancouver, British Columbia, Canada, August 1981.

[19] C. Rich and R. C. Waters. *The Programmer's Apprentice*. Addison-Wesley and ACM Press, 1990.

[20] S. Rugaber, K. Stirewalt, and L. Wills. The interleaving problem in program understanding. In *Proc. of the Second Working Conference on Reverse Engineering*, July 1995. IEEE Computer Society Press.

[21] R. Schwanke. An intelligent tool for re-engineering software modularity. In *IEEE Conf. on Software Maintenance - 1991*, pages 83-92, 1991.

[22] D. Smith, G. Kotik, and S. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Trans. on Software Engineering*, November 1985.

[23] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10(5):595-609, September 1984.

[24] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115-139, 1974.

[25] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10:352-357, 1984.

[26] L. Wills. Automated program recognition by graph parsing. Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD Thesis.

[27] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.

The Interleaving Problem in Program Understanding

Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills
College of Computing, Georgia Institute of Technology
Atlanta, Georgia 30332-0280
{spencer, kurt, linda}@cc.gatech.edu

PROGRAM
I BLE
UNDERSTANDING
T V

Abstract

One of the factors that can make a program difficult to understand is that code responsible for accomplishing more than one purpose may be woven together in a single section. We call this interleaving, and it may arise either intentionally - for example, in optimizing a program, a programmer may use some intermediate result for several purposes - or unintentionally, due to patches, quick fixes, or other hasty maintenance practices. To understand this phenomenon, we have looked at a variety of interleaving instances in actual programs and have distilled characteristic features. If the characterization proves to be robust then it will enable the design of tools for detection of interleavings and the extraction of the individual strands of computation.[1]

1 Introduction and Motivation

> Sometimes the relations between the various sections make up a maze of interwoven threads that only detailed analysis can unravel.
> - David and Mendel in *The Bach Reader.*

A program is an expression of its designers' efforts to accomplish some purpose. One aspect of understanding a program is to recreate that purpose. A particularly vexing difficulty in understanding a program is that a contiguous textual area of code can often contain fragments intended to accomplish multiple, seemingly unrelated purposes. The code fragments responsible for each of the purposes typically are delocalized and overlap rather than being composed in a simple linear sequence. We refer to these code fragments as being *interleaved.*

A trivial example is a single loop responsible for computing both the maximum element of a vector and its position. A less trivial example is a program intended to write a report that summarizes data extracted from sorted input records. The program has two purposes: computing the summary data and managing the construction of the report (headers, page breaks, page counts, etc.) In an object-oriented program, these two purposes might well be realized by two separate objects. In traditional code, however, the implementations of these functions are often interleaved, and understanding the code is significantly complicated.

We use the term *plan* to denote a description or representation of a computational structure that the designers have proposed as a way of achieving some purpose or goal in a program.[2] Plans can occur at any level of abstraction from architectural overviews to code. Interleaving expresses the merging of two or more distinct plans within some contiguous textual area of a program.

Interleaving may arise for efficiency reasons. For example, it may be more efficient to compute two related values in one place than to do so separately. Or interleaving may be the result of inadequate software maintenance, such as adding a feature locally to an existing routine rather than undertaking a thorough redesign. Or interleaving may arise as a natural by-product of expressing separate but related plans in a linear, textual medium. For example, accessors and constructors for manipulating data structures are typ-

[1] This paper discusses a number of examples of interleaving which we are making available on the World Wide Web at URL http://www.cc.gatech.edu/reverse/interleaving.html. We hope these will provide a set of challenges to other researchers interested in this problem, and we welcome additions to the set.

[2] This definition is distilled from definitions in [15, 21, 25]. Note that a plan is not necessarily stereotypical or used repeatedly; it may be novel or idiosyncratic. Following [21, 25], we reserve the term *cliché* for a plan that represents a standard, stereotypical form.

ically interleaved throughout programs written in traditional programming languages due to their procedural, rather than object-oriented structure. Regardless of why interleaving is introduced, it severely complicates understanding a program. This makes it difficult to perform tasks such as extracting reusable components, localizing the effects of maintenance changes, and migrating to object-oriented languages.

There are several reasons why interleaving is a source of difficulties. The first has to do with delocalization. Because two or more design purposes are implemented in a single segment of code, each individual code fragment responsible for a separate purpose is more spread out than it would be if it were encapsulated. Another reason why interleaving presents a problem is that it may be the result of poorly thought out maintenance activities, where the original, highly coherent structure of the system has degraded as "patches" and "quick fixes" are introduced. Finally, there may be occasions where interleaving is intentionally introduced, such as for purposes of optimization. But expressing intricate optimizations in a clean and well-documented fashion is not typically done. For all of these reasons, our ability to comprehend code containing interleaved fragments is compromised. Hopefully, we can have more success by isolating the separate concerns, understanding them individually, and only then seeing how they relate.

We are characterizing the types of interleaving that typically occur in programs in order to develop techniques for detecting and extracting interleaved, but logically cohesive plans. This paper describes our exploration, which has been based primarily on an examination of existing code from three sources:

- A Cobol database report writing system from the US Army (`IMCSRS`), which summarizes information provided as monthly updates to a master file of equipment maintenance information.
- A library of mathematical software (`SPICELIB`), written in Fortran by programmers at the Jet Propulsion Laboratory (`JPL`) for analyzing data sent back from space missions. The software performs calculations in the domain of solar system geometry.
- A program for finding the roots of functions (`ZEROIN`), presented and analyzed in [1] and [22].

The goal of the paper is to characterize the interleaving problem, relating it to existing concepts in the literature, such as delocalized plans [15], coupling [29], and redistribution of intermediate results [8, 9]. The paper presents and categorizes examples of types of interleaving to begin to map out the space of interleaving situations. It also describes how the decision to introduce interleaving into a program interacts with other design decisions. It then discusses implications for detection and extraction techniques and the types of representations needed to facilitate them.

2 Characterizing Interleaving

Interleaving can be characterized by the *delocalization* of the code for the individual plans involved, the *sharing* of some resource, and the implementation of multiple, *independent* plans in the program's overall purpose.

To illustrate these primary characteristics, we focus on an example subroutine from `SPICELIB`. Since the program contains 113 source lines of code, the full text of the program is given in the appendix. Shorter, representative excerpts will be presented and discussed in the main body of the paper.

The program, called `NPEDLN`, computes the nearest point (`PNEAR`) on an ellipsoid to a specified line. It also computes the shortest distance (`DIST`) from the ellipsoid to the line. The ellipsoid is specified by the length of its three semi-axes (`A`, `B`, and `C`), which are oriented with the x, y, and z coordinate axes. The line is specified by a point (`LINEPT`) and a direction vector (`LINEDR`).

The algorithm used to compute the nearest point checks whether the line intersects the ellipsoid (the "intercept case") and, if so, computes the intersection point. Otherwise (the "non-intercept case") it performs a more complex computation to find the line segment connecting the ellipsoid and the given line that is orthogonal to both at its endpoints; the endpoints of the computed line segment form the closest pair of points.

2.1 Delocalization

> Keep related words together. The position of the words in a sentence is the principal means of showing their relationship. Confusion and ambiguity result when words are badly placed.
> – Strunk and White in *The Elements of Style*.

`NPEDLN` has a primary goal of computing the nearest point on an ellipsoid to a specified line. It also has an orthogonal goal of ensuring that the computations involved have stable numerical behavior. A standard trick in numerical programming for achieving stability is to scale the data involved in a computation and then unscale the results. The code responsible for doing this in `NPEDLN` is scattered throughout the program's text. It is highlighted in the excerpt shown in Figure 1.

```
SUBROUTINE NPEDLN(A, B, C, LINEPT, LINEDR,
.                 PNEAR, DIST)
...
CALL UNORM ( LINEDR, UDIR, MAG )
... [error checks]
SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
SCLA   =  A / SCALE
SCLB   =  B / SCALE
SCLC   =  C / SCALE
... [error checks]
SCLPT(1)  =  LINEPT(1) / SCALE
SCLPT(2)  =  LINEPT(2) / SCALE
SCLPT(3)  =  LINEPT(3) / SCALE
CALL VMINUS ( UDIR, OPPDIR )
CALL SURFPT ( SCLPT, UDIR, SCLA, SCLB,
.             SCLC,PT(1,1), FOUND(1))
CALL SURFPT ( SCLPT, OPPDIR,SCLA, SCLB,
.             SCLC, PT(1,2), FOUND(2))
... [checking for intersection of the
     line with the ellipsoid]
    IF ( FOUND(I) ) THEN
       DIST  =  0.0D0
       CALL VSCL   (SCALE, PNEAR, PNEAR)
       ...
       RETURN
    END IF
... [handling the non-intercept case]
CALL VSCL ( SCALE,  PNEAR,  PNEAR )
DIST   =  SCALE * DIST
...
RETURN
END
```

Figure 1: **Portions of the NPEDLN Fortran program. Shaded regions highlight the lines of code responsible for scaling and unscaling.**

The delocalized nature of this "scale-unscale" plan makes it difficult to gather together all the pieces involved for consistent maintenance. It also gets in the way of understanding the rest of the code, since it provides distracting details that must be filtered out. *Delocalization* is one of the key characteristics of interleaving: one or more components of a plan are spatially separated from other components by the components of other plans with which they are interleaved. Letovsky and Soloway's cognitive study [15] shows the deleterious effects of delocalization on comprehension and maintenance.

The "scale-unscale" pattern is a simple example of a more general delocalized plan, ("transform-untransform") which is frequently interleaved with computations in SPICELIB. The general form, which we refer to as *reformulation wrappers*, is used to transform one problem into another that is simpler to solve, and then to transfer the solution back to the original situation. Some examples of reformulation wrappers in SPICELIB are: reducing a three-dimensional geometry problem to a two-dimensional one and mapping an ellipsoid to the unit sphere to make it easier to solve three-dimensional intersection problems.

Delocalization may occur for a variety of different reasons. One is that there may be an inherently non-local relationship between the components of the plan, as is the case with reformulation wrappers, which makes the spatial separation necessary. Another reason is that the intermediate results of part of a plan may be shared with another plan, causing the plans to overlap and their steps to be shuffled together; the steps of one plan separate those of the other. For example, in Figure 1, part of the scale plan (computing the scaling factor) is separated from the rest of the plan (dividing by the scaling factor) in all scalings, except the scaling of A. This allows the scaling factor to be computed once and the result reused. (The role of sharing in interleaving is discussed more extensively in Section 2.2.)

Realizing that a reformulation wrapper or some other delocalized plan is interleaved with a particular computation can help prevent severe comprehension failures during maintenance [15]. It can also help detect when the delocalized plan is incomplete, as it was in an earlier version of our example subroutine whose modification history includes the following correction:

```
C- SPICELIB Version 1.2.0, 25-NOV-1992 (NJB)
C Bug fix: in the intercept case, PNEAR is now
C properly re-scaled prior to output.  Formerly,
C it was returned without having been re-scaled.
```

2.2 Resource Sharing

> The complexity of multifunctioning elements can sometimes turn data graphics into visual puzzles, crypto-graphical mysteries for the viewer to decode.
>
> – Edward Tufte in *The Visual Display of Quantitative Information.*

In addition to computing the nearest point on the ellipsoid to the line, NPEDLN computes the shortest distance between the line and the ellipsoid. This additional output (DIST) is convenient to construct, based on intermediate results obtained while computing the primary output (PNEAR). This is illustrated in Figure 2. The shaded portions of the code shown are shared between the two computations for PNEAR and DIST. (The computation of DIST using VDIST is actually the last computation performed by the subroutine NPELPT, which NPEDLN calls; we have pulled this computation out of NPELPT for clarity of presentation.)

The sharing of some resource is characteristic of intentional interleaving. When interleaving is introduced into a program, there is normally some implicit relationship between the interleaved plans, motivating the designer to choose to interleave them. In this case, interleaved plans share some common resource – intermediate data results. Their implementations

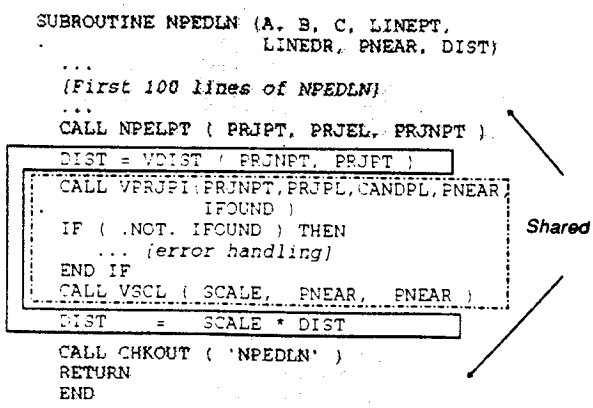

Figure 2: **Portions of NPEDLN, highlighting two overlapping computations.**

overlap in that a single structural element contributes to multiple goals.

The sharing of the results of some subcomputation in the implementation of two distinct higher level operations is termed *redistribution of intermediate results* by Hall [8, 9]. More specifically, redistribution is a class of function sharing optimizations which are implemented simply by tapping into the dataflow from some value producer and feeding it to an additional target consumer, essentially introducing fanout in dataflow. Redistribution covers a wide range of common types of function sharing optimizations, including common subexpression elimination and generalized loop fusion. Hall developed an automated technique for redistributing results for use in optimizing code generated from general-purpose reusable software components. We are interested in "undoing" these types of optimizations, and we can use the redistribution concept to capture forms of interleaving in which the resources shared are *data* values.

The commonality between interleaved plans might be in the form of other shared resources besides data entities, such as control structures, lexical module structures, and names.

Control Coupling. Control conditions may be redistributed just as data values are. The use of control flags allows control conditions to be determined once but used to affect execution at more than one location in the program. In NPEDLN, for example, SURFPT is called to compute the intersection of the line with the ellipsoid. This routine returns a control flag FOUND, indicating whether or not the intersection exists. This flag is then used outside of SURFPT to control whether the intercept or non-intercept case is to be handled as is shown below.

```
      CALL SURFPT ( SCLPT, UDIR,   SCLA, SCLB,
     .              SCLC, PT(1,1), FOUND(1) )
      CALL SURFPT ( SCLPT, OPPDIR, SCLA, SCLB,
     .              SCLC, PT(1,2), FOUND(2) )
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
           ... [handling the intercept case]
           RETURN
         END IF
50001 CONTINUE
C     Getting here means the line doesn't
C     intersect the ellipsoid.
      ... [handling the non-intercept case]
      RETURN
      END
```

The use of control flags is a special form of *control coupling*: "any connection between two modules that communicates elements of control [29]," typically in the form of function codes, flags, or switches [16]. This sharing of control information between two modules increases the complexity of the code, complicating comprehension and maintenance.

Content coupling. Another form of resource sharing occurs when the lexical structure of a module is shared among several related functional components. The entire contents of a module may be lexically included in another or there may be partial overlap of modules. This is called *content coupling* [29] – "some or all of the contents of one module are included in the contents of another" – and often manifests itself in the form of a multiple-entry module. Content coupling makes it difficult to independently modify or maintain the individual functions.

Name Sharing. A simple form of sharing is the use of the same variable name for two different purposes. This can lead to incorrect assumptions about the relationship between subcomputations within a program.

In general, the difficulty that resource sharing introduces is that it causes ambiguity in interpreting the purpose of program pieces. This can lead to incorrect assumptions about what effect changes will have, since the maintainer might be focusing on only one of the actual uses of the resource (variable, value, control flag, data structure slot, etc.).

```
SUBROUTINE NPEDLN (A, B, C, LINEPT,
                   LINEDR, PNEAR, DIST)
  ...
  CALL UNORM ( LINEDR, UDIR, MAG )
  IF ( MAG .EQ. 0 ) THEN
     CALL SETMSG('Dir. is zero vector.')
     ... [error signaling]
     RETURN
  ELSE IF (    ( A .LE. 0.D0 )
     .OR.( B .LE. 0.D0 )
     .OR.( C .LE. 0.D0 ))   THEN
     CALL SETMSG('Semi-axes:A=#,B=#,C=#.')
     ... [error signaling]
     RETURN
  END IF
  ... [scaling]
  IF (    ( SCLA**2  .LE.  0.D0 )
     .OR.( SCLB**2  .LE.  0.D0 )
     .OR.( SCLC**2  .LE.  0.D0 ))   THEN
     CALL SETMSG('Too small:A=#,B=#,C=#.')
     ... [error signaling]
     RETURN
  END IF
  ... [scaling]
  ... [handling the intercept case]
  ... [handling the non-intercept case]
  IF ( .NOT. XFOUND ) THEN
    CALL SETMSG ('Cand ellipse not found.')
    ... [error signaling]
    RETURN
  END IF
  ...
```

Figure 3: **Portions of NPEDLN, highlighting code responsible for checking preconditions.**

2.3 Independence

> ... You know they're going to get together ..., but it's fun to watch how they keep missing... It's emotionally satisfying because you get that cathartic moment at the end... It's intellectually satisfying because the plot's always twisting in on itself. – Susan Seidelman, director, on romantic comedy in *American Cinema*.

While interleaving is introduced to take advantage of commonalities, the flip side of the coin is that the interleaved plans each have a distinct purpose. One implication of this is that the decision to interleave the plans can, in principle, always be undone. This usually requires copying of common code to eliminate resource sharing, resulting in an equivalent, but possibly less efficient, program.

The NPEDLN program, for instance, computes the nearest point on an ellipsoid to a specified line. It also checks a set of preconditions which must be true for the computation to be carried out correctly: the line must not be a zero vector, the ellipsoid's semi-axes must have positive length and must be large enough to be scalable, and the ellipsoid must not be too flat or needle-shaped. Figure 3 shows an excerpt of NPEDLN, highlighting the precondition checking code.

In principle, all these checks can be performed beforehand, pulling them out of the midst of the primary computation and having them preface it. This would require duplicating almost the entire program, since the precondition checking makes heavy use of intermediate results computed throughout the program.

Although interleaving is necessary for efficiency, it obscures the independence of the components involved. Ironically, this hinders activities for making the code more efficient and reusable in the long run, such as parallelization and objectivization of the code.

3 Implications for Representation, Detection, and Extraction

We have identified the characteristics of interleaving that make it troublesome in understanding programs. We can now consider the implications of what we have learned from our empirical study. In particular, we are interested in understanding how the decision to interleave interacts with other design decisions; what types of program representations will facilitate the detection of interleaving decisions and the extraction of interleaved components; and how can interleaved components be automatically detected and extracted from code?

3.1 Interactions with Other Decisions

Intentional interleaving is one of several ways in which design decisions are elaborated in code [22]. More familiar examples include the decomposition of a complex concept into its constituents (aggregation), the processing of a general problem in terms of various individual cases (specialization), the elaboration of a generic concept as a more concrete artifact (instantiation), and the modeling of one structure in terms of another (representation). The converse of interleaving is encapsulation, where a designer intentionally delineates several distinct design elements behind some sort of barrier, such as provided by the PACKAGE mechanism in Ada.

The history of the design of a program can be viewed as a network of artifacts [2] where a connection between two artifacts indicates that one of them is the refinement of the other that results from one of the kinds of design decisions listed above. In this view, interleaving is indicated by a network node with inputs from *two or more* other nodes. It may well be the case, however, that by tracing further back in the history some common ancestor of the two inputs can be found. For example, NPEDLN uses a control flag to indicate whether to handle the intercept or non-intercept case. In looking at the code for the subprogram, we see only instances of interleaved code fragments. In reviewing the design history, however, we may find

that what is more fundamental is that a decision has been made to specialize the code into these two cases, that the flag distinguishes the two cases, and the interleaving is really just a manifestation of the higher level specialization decision.

In the case of the loop that is being used to compute both the maximum element in a vector and its index, there may have been an aggregation decision in the design history that justifies why both values are needed and only later an interleaving optimization to save some loop overhead.

In instances where a more fundamental decision has provoked a later interleaving, not only is the rationale connecting the two decisions usually lost, but we end up viewing the code in delocalized form when we really would like to see a factored version.

We postulate that interleaving often co-occurs with certain other design decisions. If further empirical study confirms this, then interleaving removal could be seen as *enabling* the reversal of other decisions or, dually, that certain design decisions enable interleaving by providing opportunities for resource sharing.

3.2 Requirements for Representations

Three key characteristics of interleaving are: delocalization, resource sharing, and independence. Delocalization results from having to serialize the components of two or more separate plans. This total ordering is necessary due to the lack of support for concurrency in most high level programming languages. It follows then that in order to *undo* delocalization a representation must impose a partial rather than a total execution ordering on the components of plans.

The partial execution ordering requirement suggests that some form of graphical representation is appropriate. Graph representations naturally express a partial execution ordering via implicit concurrency and explicit transfer of control and data. Since there are a number of such representations to choose from, we narrow the search space by noting that:

1. independent plans must be localized as much as possible, with no explicit ordering between them,
2. sharing must be detectable (shared resources should explicitly flow from one plan to another); similarly if two plans p_1, p_2 both share a resource provided by a plan p_3 then p_1 and p_2 should appear in the graph as siblings with a common ancestor p_3), and
3. the representation must support multiple views of the program as the interaction of plans at various levels of abstraction, since interleaving may occur at any level of abstraction.

An existing formalism that meets these criteria is Rich's *Plan Calculus* [19, 20], which was developed and used in the Programmer's Apprentice [21] project. A plan in the Plan Calculus is encoded as a graphical depiction of the plan's structural parts and the constraints (e.g., data and control flow connections) between them. This diagrammatic notation is complemented with an axiomatized description of the plan which defines its formal semantics. This allows us to develop correctness preserving transformations to extract interleaved plans. The Plan Calculus also provides a mechanism, called *overlays*, for representing correspondences and relationships between pairs of plans (e.g., implementation and optimization relationships). This enables the viewing of plans at multiple levels of abstraction. Overlays also support a very general notion of plan composition which takes into account resource sharing at all levels of abstraction by allowing overlapping points of view.

3.3 Support for Detection and Extraction

> Do not be afraid to seize whatever you have written and cut it to ribbons; it can always be restored to its original condition in the morning, if that course seems best. – Strunk & White in *Elements of Style*.

In order to develop tools for detecting and extracting interleaving, it is helpful to consider how interleaving manifests itself in source code. There are three useful, orthogonal dimensions. These form a possible design space of solutions to the interleaving problem and can help relate existing techniques that might be applicable. One dimension is the *scope* of the interleaving, which can range from intraprocedural to interprocedural to object to architectural.

Another dimension is the *structural mechanism* for providing interleaving, which may be naming, control, data, or protocol (i.e., global constraints, such as maintaining stack discipline or synchronization mechanisms for cooperating processes). For example, the use of control flags is a control-based mechanism for interleaving with interprocedural scope. The common iteration construct involved in loop fusion is another control-based mechanism, but the interleaving has intraprocedural scope. Reformulation wrappers use a protocol mechanism, usually at the intraprocedural level, but they can have interprocedural scope. Multiple-inheritance is an example of a data-centered interleaving mechanism with object scope.

The third dimension is the *familiarity* of the plans interleaved: are they clichés (i.e., stereotypical, frequently used plans) or are they unfamiliar plans (i.e., novel, idiosyncratic, or not used repeatedly)?

When what is interleaved is *familiar* (i.e., a cliché), cliché recognition (e.g., [10, 12, 13, 14, 18, 28]) is a useful detection mechanism.[3] In fact, most recognition systems deal explicitly with the recognition of clichés that are interleaved in specific ways with unrecognizable code or other clichés. One of the key features of GRASPR [28], for instance, is its ability to deal with delocalization and redistribution-type function sharing optimizations.

KBEmacs [21, 26] uses a simple, special-purpose recognition strategy to segment loops within programs. This is based on detecting coarse patterns of data and control flow at the procedural level that are indicative of common ways of constructing, augmenting, and interleaving iterative computations. For example, KBEmacs looks for minimal sections of a loop body which have data flow feeding back only to themselves. This decomposition enables a powerful form of abstraction, called *temporal abstraction*, which views iterative computations as compositions of operations on sequences of values. The recognition and temporal abstraction of iteration clichés is similarly used in GRASPR to enable it to deal with generalized loop fusion forms of interleaving (loop fusion is viewed as redistribution of sequences of values and treated as any other redistribution optimization) [28].

Existing cliché recognition systems tend to deal with interleaving involving *data* and *control* mechanisms. Domain-based clustering, as explored by DM-TAO in the DESIRE system [3], focuses on *naming* mechanisms, by keying in on the patterns of linguistic idioms used in the program, which suggest the manifestations of domain concepts.

When *unfamiliar* plans are interleaved, other, non-recognition-based methods of delineation are needed. For example, slicing [27, 17] is a widely-used technique for localizing functional components by tracing through data dependencies within the procedural scope. Cluster analysis [3, 11, 23, 24] is used to group related sections of code, based on the detection of shared uses of global data, control paths, and names. However, clustering techniques can only provide limited assistance by roughly delineating possible locations of functionally cohesive components. Another technique, called "potpourri module detection" [5], detects modules that provide more than one independent service by looking for multiple proper subgraphs in an entity-to-entity interconnection graph. These graphs show dependencies among global entities within a single module. Presumably, the independent services reflect separate plans in the code.

Research into automating data encapsulation has recently provided mechanisms for hypothesizing possible locations of data plans at the *object* scope. For example, Bowdidge and Griswold [4] use an extended data flow graph representation, called a star diagram, to help human users see all the uses of a particular data structure and to detect frequently occurring computations that are good candidates for abstract functions. Techniques have also been developed within the RE^2 project [6, 7], for identifying candidate abstract data types and their associated modules, based on the call graph and dominance relations. Further research is required to develop techniques for extracting objects from pieces of data that have not already been aggregated in programmer-defined data structures. For example, detecting multiple pieces of data that are always used together might suggest candidates for data aggregation (as for example, in NPEDLN, where the input parameters A, B, and C are used as a tuple representing an ellipsoid, and the outputs PNEAR and DIST represent a pair of results related by interleaved, highly overlapping plans).

Interleaving at the scope of objects and architectures and/or involving global protocol mechanisms is not yet well understood. Consequently, few mechanisms for detection and extraction currently exist in these areas. We believe that more and more knowledge of the domain will be required to detect interleaving as the scope becomes more global in nature. Also, a key open issue is to what extent techniques for detecting and extracting interleaved plans must rely on the familiarity of the plans involved; how far can we go with non-recognition-based techniques?

4 Conclusion

This paper characterizes interleaving, a particularly troublesome feature of programs which makes them difficult to understand. We offer the following definition:

> *Interleaving* expresses the merging of two or more distinct plans within some contiguous textual area of a program. Interleaving can be characterized by the *delocalization* of the code for the individual plans involved, the *sharing* of some resource, and the implementation of multiple, *independent* plans in the program's overall purpose.

Whether this characterization is robust is an issue for future empirical studies. We plan to study the

[3] Recognition as a program understanding technique deals with clichés, not plans in general. Only clichéd plans can be recognized, since recognition implies noticing something that is familiar.

frequency of occurrence of the various types of interleaving we have identified in our example programs. This may lead to better complexity metrics for determining the maintainability and comprehensibility of programs. Ultimately, designing tools for detection and extraction will be the true test of the usefulness of this characterization.

Acknowledgments

Support for this research has been provided by ARPA under order number A870, contract number NAG 2-890. We are grateful to JPL's NAIF group for enabling our study of their SPICELIB software. We also benefited from insightful discussions with Michael Lowry at Nasa Ames Research Center concerning this study and interesting future directions. We would also like to thank the reviewers for their helpful comments.

References

[1] V.R. Basili and H.D. Mills. Understanding and documenting programs. *IEEE Trans. on Software Engineering*, 8(3):270–283, May 1982.

[2] I. Baxter. Design maintenance systems. *Comm. of the ACM*, 35(4), April 1992.

[3] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Comm. of the ACM*, 37(5):72–83, May 1994.

[4] R. Bowdidge and W. Griswold. Automated support for encapsulating abstract data types. In *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 97–110, New Orleans, Dec. 1994.

[5] F. Calliss and B. Cornelius. Potpourri module detection. In *IEEE Conf. on Software Maintenance - 1990*, pages 46–51, San Diego, CA, November 1990. IEEE Computer Society Press.

[6] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data types. In *Proc. of the First Working Conference on Reverse Engineering*, pages 73–82. Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[7] A. Cimitile, M. Tortorella, and M. Munro. Program comprehension through the identification of abstract data types. In *Proc. 3rd Workshop on Program Comprehension*, pages 12–19, Washington, D.C., November 1994. IEEE Computer Society Press.

[8] R. Hall. Program improvement by automatic redistribution of intermediate results. Technical Report 1251, MIT Artificial Intelligence Lab., February 1990. PhD.

[9] R. Hall. Program improvement by automatic redistribution of intermediate results: An overview. In M. Lowry and R. McCartney, editors, *Automating Software Design*. AAAI Press, Menlo Park, CA, 1991.

[10] J. Hartman. Automatic control understanding for natural programs. Technical Report AI 91-161, University of Texas at Austin, 1991. PhD thesis.

[11] D. Hutchens and V. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. on Software Engineering*, 11(8), August 1985.

[12] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[13] W. Kozaczynski and J.Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1(1):61–78, March 1994.

[14] S. Letovsky. Plan analysis of programs. Research Report 662, Yale University, December 1988. PhD.

[15] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3), 1986.

[16] G. Myers. *Reliable Software through Composite Design*. Petrocelli Charter, 1975.

[17] J.Q. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Comm. of the ACM*, 37(5):50–57, May 1994.

[18] A. Quilici. A memory-based approach to recognizing programming plans. *Comm. of the ACM*, 37(5):84–93, May 1994.

[19] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981.

[20] C. Rich. Inspection methods in programming. Technical Report 604, MIT Artificial Intelligence Lab., June 1981. PhD thesis.

[21] C. Rich and R. C. Waters. *The Programmer's Apprentice*. Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.

[22] S. Rugaber, S. Ornburn, and R. LeBlanc. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[23] R. Schwanke. An intelligent tool for re-engineering software modularity. In *IEEE Conf. on Software Maintenance - 1991*, pages 83–92, 1991.

[24] R. Schwanke, R. Altucher, and M. Platoff. Discovering, visualizing, and controlling software structure. In *Proc. 5th Int. Workshop on Software Specs. and Design*, pages 147–150, Pittsburgh, PA, 1989.

[25] P. Selfridge, R. Waters, and E. Chikofsky. Challenges to the field of reverse engineering - A position paper. In *Proc. of the First Working Conference on Reverse Engineering*, pages 144–150, Baltimore, Maryland, May 1993. IEEE Computer Society Press.

[26] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[27] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10:352–357, 1984.

[28] L. Wills. Automated program recognition by graph parsing. Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD Thesis.

[29] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1979.

5 Appendix: NPEDLN

```
C$ Nearest point on ellipsoid to line.
SUBROUTINE NPEDLN(A,B,C,LINEPT,LINEDR,PNEAR,DIST)
  INTEGER               UBEL
  PARAMETER           ( UBEL = 9 )
  INTEGER               UBPL
  PARAMETER           ( UBPL = 4 )
  DOUBLE PRECISION      A
  DOUBLE PRECISION      B
  DOUBLE PRECISION      C
  DOUBLE PRECISION      LINEPT ( 3 )
  DOUBLE PRECISION      LINEDR ( 3 )
  DOUBLE PRECISION      PNEAR  ( 3 )
  DOUBLE PRECISION      DIST
  LOGICAL               RETURN
  DOUBLE PRECISION      CANDPL ( UBPL )
  DOUBLE PRECISION      CAND   ( UBEL )
  DOUBLE PRECISION      OPPDIR (    3 )
  DOUBLE PRECISION      PRJPL  ( UBPL )
  DOUBLE PRECISION      MAG
  DOUBLE PRECISION      NORMAL (3 )
  DOUBLE PRECISION      PRJEL  ( UBEL )
  DOUBLE PRECISION      PRJPT  (3 )
  DOUBLE PRECISION      PRJNPT (    3 )
  DOUBLE PRECISION      PT     ( 3, 2 )
  DOUBLE PRECISION      SCALE
  DOUBLE PRECISION      SCLA
  DOUBLE PRECISION      SCLB
  DOUBLE PRECISION      SCLC
  DOUBLE PRECISION      SCLPT  (    3 )
  DOUBLE PRECISION      UDIR   (    3 )
  INTEGER               I
  LOGICAL               FOUND  (    2 )
  LOGICAL               IFOUND
  LOGICAL               XFOUND
  IF ( RETURN () ) THEN
     RETURN
  ELSE
     CALL CHKIN ( 'NPEDLN' )
  END IF
  CALL UNORM ( LINEDR, UDIR, MAG )
  IF ( MAG .EQ. 0 ) THEN
     CALL SETMSG('Direction is zero vector.')
     CALL SIGERR('SPICE(ZEROVECTOR)'        )
     CALL CHKOUT('NPEDLN'                   )
     RETURN
   ELSE IF (( A .LE. 0.D0 )
            .OR.   ( B .LE. 0.D0 )
            .OR.   ( C .LE. 0.D0 )   )   THEN
     CALL SETMSG ('Semi-axes: A=#,B=#,C=#.')
     CALL ERRDP  ('#', A                    )
     CALL ERRDP  ('#', B                    )
     CALL ERRDP  ('#', C                    )
     CALL SIGERR ('SPICE(INVALIDAXISLENGTH)')
     CALL CHKOUT ('NPEDLN'                  )
     RETURN
   END IF
C Scale the semi-axes lengths for better numerical
C behavior.  If squaring any of the scaled lengths
C causes it to underflow to zero, signal an error.
C Otherwise scale the point on the input line too.
  SCALE  =  MAX ( DABS(A), DABS(B), DABS(C) )
  SCLA   =  A / SCALE
  SCLB   =  B / SCALE
  SCLC   =  C / SCALE
  IF (( SCLA**2  .LE.  0.D0 )
      .OR.  ( SCLB**2  .LE.  0.D0 )
      .OR.  ( SCLC**2  .LE.  0.D0 )   )   THEN
     CALL SETMSG ('Axis too small: A=#,B=#,C=#.')
     CALL ERRDP  ('#', A                        )
     CALL ERRDP  ('#', B                        )
     CALL ERRDP  ('#', C                        )
     CALL SIGERR ('SPICE(DEGENERATECASE)'       )
     CALL CHKOUT ('NPEDLN'                      )
     RETURN
  END IF
  SCLPT(1)  =  LINEPT(1) / SCALE
  SCLPT(2)  =  LINEPT(2) / SCALE
  SCLPT(3)  =  LINEPT(3) / SCALE
C Hand off the intersection case to SURFPT.
C SURFPT determines whether rays intersect a body,
C so we treat the line as a pair of rays.
  CALL VMINUS(UDIR, OPPDIR)
  CALL SURFPT(SCLPT, UDIR,   SCLA, SCLB,
              SCLC, PT(1,1), FOUND(1))
```

```
      CALL SURFPT(SCLPT, OPPDIR, SCLA, SCLB,
     .            SCLC, PT(1,2), FOUND(2))
      DO 50001
     .   I = 1, 2
         IF ( FOUND(I) ) THEN
            DIST  =  0.0D0
            CALL VEQU   ( PT(1,I),  PNEAR         )
            CALL VSCL   ( SCALE,    PNEAR,  PNEAR )
            CALL CHKOUT ( 'NPEDLN'                )
            RETURN
         END IF
50001 CONTINUE
C Getting here means the line doesn't intersect
C the ellipsoid. Find the candidate ellipse CAND.
C NORMAL is a normal vector to the plane
C containing the candidate ellipse. Mathematically
C the ellipse must exist; it's the intersection of
C an ellipsoid centered at the origin and a plane
C containing the origin.  Only numerical problems
C can prevent the intersection from being found.
      NORMAL(1)  =  UDIR(1) / SCLA**2
      NORMAL(2)  =  UDIR(2) / SCLB**2
      NORMAL(3)  =  UDIR(3) / SCLC**2
      CALL NVC2PL ( NORMAL, 0.D0, CANDPL )
      CALL INEDPL (SCLA,SCLB,SCLC,CANDPL,CAND,XFOUND)
      IF ( .NOT. XFOUND ) THEN
         CALL SETMSG ( 'Candidate ellipse not found.')
         CALL SIGERR ( 'SPICE(DEGENERATECASE)'       )
         CALL CHKOUT ( 'NPEDLN'                      )
         RETURN
      END IF
C Project the candidate ellipse onto a plane
C orthogonal to the line.  We'll call the plane
C PRJPL and the projected ellipse PRJEL.
      CALL NVC2PL ( UDIR,  0.D0,   PRJPL )
      CALL PJELPL ( CAND,  PRJPL,  PRJEL )
C Find the point on the line lying in the project-
C ion plane, and then find the near point PRJNPT
C on the projected ellipse.  Here PRJPT is the
C point on the line lying in the projection plane.
C The distance between PRJPT and PRJNPT is DIST.
      CALL VPRJP  ( SCLPT,   PRJPL,  PRJPT         )
      CALL NPELPT ( PRJPT,   PRJEL,  PRJNPT )
      DIST = VDIST ( PRJNPT, PRJPT )
C Find the near point PNEAR on the ellipsoid by
C taking the inverse orthogonal projection of
C PRJNPT; this is the point on the candidate
C ellipse that projects to PRJNPT.  The output
C DIST was computed in step 3 and needs only to be
C re-scaled. The inverse projection of PNEAR ought
C to exist, but may not be calculable due to nu-
C merical problems (this can only happen when the
C ellipsoid is extremely flat or needle-shaped).
      CALL VPRJPI(PRJNPT,PRJPL, CANDPL, PNEAR, IFOUND)
      IF ( .NOT. IFOUND ) THEN
         CALL SETMSG ('Inverse projection not found.')
         CALL SIGERR ('SPICE(DEGENERATECASE)'       )
         CALL CHKOUT ('NPEDLN'                      )
         RETURN
      END IF
C Undo the scaling.
      CALL VSCL ( SCALE,  PNEAR,  PNEAR )
      DIST   =   SCALE * DIST
      CALL CHKOUT ( 'NPEDLN' )
      RETURN
      END
C ----------------------------------------------
C Descriptions of subroutines called by NPEDLN:
C CHKIN  Module Check In (error handling).
C UNORM  Normalize double precision 3-vector.
C SETMSG Set Long Error Message.
C SIGERR Signal Error Condition.
C CHKOUT Module Check Out (error handling).
C ERRDP  Insert DP Number into Error Message Text.
C VMINUS Negate a double precision 3-D vector.
C SURFPT Find intersection of vector w/ ellipsoid.
C VEQU   Make one DP 3-D vector equal to another.
C VSCL   Vector scaling, 3 dimensions.
C NVC2PL Make plane from normal and constant.
C INEDPL Intersection of ellipsoid and plane.
C PJELPL Project ellipse onto plane, orthogonally.
C VPRJP  Project a vector onto plane orthogonally.
C NPELPT Find nearest point on ellipse to point.
C VPRJPI Vector projection onto plane, inverted.
C ----------------------------------------------
C Descriptions of variables used by NPEDLN:
C A       Length of semi-axis in the x direction.
C B       Length of semi-axis in the y direction.
C C       Length of semi-axis in the z direction.
C LINEPT  Point on input line.
C LINEDR  Direction vector of input line.
C PNEAR   Nearest point on ellipsoid to line.
C DIST    Distance of ellipsoid from line.
C UBEL    Upper bound of array containing ellipse.
C UBPL    Upper bound of array containing plane.
C PT      Intersection point of line & ellipsoid.
C CAND    Candidate ellipse.
C CANDPL  Plane containing candidate ellipse.
C NORMAL  Normal to the candidate plane CANDPL.
C UDIR    Unitized line direction vector.
C MAG     Magnitude of line direction vector.
C OPPDIR  Vector in direction opposite to UDIR.
C PRJPL   Projection plane, which the candidate
C         ellipse is projected onto to yield PRJEL.
C PRJEL   Projection of the candidate ellipse
C         CAND onto the projection plane PRJEL.
C PRJPT   Projection of line point.
C PRJNPT  Nearest point on projected ellipse to
C         projection of line point.
C SCALE   Scaling factor.
```